Scabbard: An Exploratory Study on Hardware Aware Design Choices of Learning with Rounding-based Key Encapsulation Mechanisms

SUPARNA KUNDU and QUINTEN NORGA, COSIC, KU Leuven, Belgium
ANGSHUMAN KARMAKAR, IIT Kanpur, India
SHREYA GANGOPADHYAY, IIT Kharagpur, India
JOSE MARIA BERMUDO MERA, PQShield, Oxford, UK
INGRID VERBAUWHEDE, COSIC, KU Leuven, Belgium

Recently, the construction of cryptographic schemes based on hard lattice problems has gained immense popularity. Apart from being quantum resistant, lattice-based cryptography allows a wide range of variations in the underlying hard problem. As cryptographic schemes can work in different environments under different operational constraints such as memory footprint, silicon area, efficiency, power requirement, etc., such variations in the underlying hard problem are very useful for designers to construct different cryptographic schemes. In this work, we explore various design choices of lattice-based cryptography and their impact on performance in the real world. In particular, we propose a suite of key-encapsulation mechanisms based on the learning with rounding problem with a focus on improving different performance aspects of lattice-based cryptography. Our suite consists of three schemes. Our first scheme is Florete, which is designed for efficiency. The second scheme is Espada, which is aimed at improving parallelization, flexibility, and memory footprint. The last scheme is Sable, which can be considered an improved version in terms of key sizes and parameters of the Saber key-encapsulation mechanism, one of the finalists in the National Institute of Standards and Technology's post-quantum standardization procedure. In this work, we have described our design rationale behind each scheme.

Further, to demonstrate the justification of our design decisions, we have provided software and hardware implementations. Our results show Florete is faster than most state-of-the-art KEMs on software platforms. For example, the key-generation algorithm of high-security version Florete outperforms the National Institute of Standards and Technology's standard Kyber by 47%, the Federal Office for Information Security's standard Frodo by 99%, and Saber by 57% on the ARM Cortex-M4 platform. Similarly, in hardware, Florete outperforms Frodo and NTRU Prime for all KEM operations. The scheme Espada requires less memory and area than the implementation of most state-of-the-art schemes. For example, the encapsulation algorithm of high-security version Espada uses 30% less stack memory than Kyber, 57% less stack memory than Frodo, and 67% less stack memory than Saber on the ARM Cortex-M4 platform. The implementations of Sable maintain a trade-off between Florete and Espada regarding software performance and memory requirements. Sable outperforms Saber at least by 6% and Frodo by 99%. Through an efficient polynomial multiplier design, which exploits the small secret size, Sable outperforms most state-of-the-art KEMs, including Saber, Frodo, and NTRU Prime. The implementations of Sable that use number theoretic transform-based polynomial multiplication (SableNTT) surpass all the state-of-the-art schemes in performance, which are optimized for speed on the Context M4 platform. The performance benefit of SableNTT against Kyber lies in between $7 - 29\%$, $2 - 13\%$ for Saber, and around 99% for Frodo.

CCS Concepts: • **Security and privacy → Public key encryption**; Hardware-based security protocols.

Additional Key Words and Phrases: Post-quantum cryptography, Lattice-based cryptography, Learning with rounding, Key-encapsulation mechanism, Software implementations, AVX2, Cortex-M4, Hardware implementations, FPGA

Authors' addresses: Suparna Kundu, Suparna.Kundu@esat.kuleuven.be; Quinten Norga, Quinten.Norga@esat.kuleuven.be, COSIC, KU Leuven, Kasteelpark Arenberg 10, Bus 2452, B-3001 Leuven-Heverlee, Belgium; Angshuman Karmakar, angshuman@cse.iitk.ac.in, IIT Kanpur, India; Shreya Gangopadhyay, gangopadhyay.shreya09@gmail.com, IIT Kharagpur, India; Jose Maria Bermudo Mera, josebmera@gmail.com, PQShield, Oxford, UK; Ingrid Verbauwhede, Ingrid.Verbauwhede@esat.kuleuven.be, COSIC, KU Leuven, Kasteelpark Arenberg 10, Bus 2452, B-3001 Leuven-Heverlee, Belgium.

1 INTRODUCTION

Lattice-based cryptography has been one of the most discussed topics in public-key cryptography (PKC) for the past several years. Apart from being resistant to quantum attacks and hence a possible alternative for integer-factorization (IF) and discrete-log problem (DLP)-based cryptographic constructions, lattice-based cryptographic constructions are relatively simpler. Moreover, compared to IF and DLP, lattices offer lots of variations of underlying hard problems. This provides cryptographic designers with lots of maneuvering space to explore different designs to optimize and curate their cryptographic constructions for different applications. For example, from Ajtai's short-integer solution (SIS) [3] and Hoffstein et al.'s NTRU [59] in 1996 to Regev's learning with errors (LWE) [98] in 2005 and its subsequent variations such as Ring-LWE [81], Module-LWE [78], learning with rounding [8, 13], and recently discovered PLWE [99], CLWE [31], etc., the choice of computationally hard problems to design cryptographic schemes is in galore. Nevertheless, lattice-based cryptography has not always been the most preferred choice for cryptographers. IF and DLP-based cryptography, which were invented a couple of decades earlier, had already well-established themselves in the existing public-key infrastructure. Due to lots of research on their implementation, side-channel security, cryptanalysis, etc., their theoretical and implementation aspects were well understood. Therefore, there was little incentive for theorists and practitioners alike to replace these classical cryptosystems with lattice-based cryptography even though Shor's [93, 101] algorithm and its detrimental effect on IF and DLP-based cryptography was known since 1994. This happened partly because quantum computing research was mostly restricted to the realm of theory, and there was skepticism about its physical existence in the future.

However, as the research on developing large-scale quantum computers gained momentum, the future of IF and DLP-based cryptosystems as mainstream PKC algorithms started looking bleak proportionately. Due to the recent advancements in the field of quantum computing, the adverse effects of quantum computers on our existing public-key infrastructure have become too hard to ignore further. Although the research in quantum-resistant PKC or post-quantum cryptography started a couple of decades ago, the watershed moment in the process of transitioning from classical PKC to PQC is the National Institute of Standards and Technology's (NIST) conclusion of a long and multi-staged standardization procedure [4] in 2022. NIST standardized PQC primitives such as public-key encryption (PKE) or key-encapsulation mechanism (KEM) Kyber [27], and digital signature schemes CRYSTALS-Dilithium [43], FALCON [46], and SPHINCS+ [11].

During NIST's standardization process, the cryptographic community witnessed many innovations in the design and implementation of PQC. Such as the introduction of module lattices [78] instead of more traditional standard [98] or ideal [81] lattices as a trade-off between speed and security, usage of central binomial distributions [7] instead of discrete Gaussian distribution for protection against potential side-channel attacks, just-in-time matrix generation in module lattices [68] and improvements in polynomial multiplications [24, 37, 62, 82] algorithms to improve efficiency, the introduction of error-correcting codes [23] to reduce the decryption failure rates of lattice-based cryptography, etc. These different designs went through a thorough and rigorous evaluation. For example, the non constant-time behavior of error-correcting codes was found to be highly vulnerable to side-channel attacks; similarly, the sparse distributions used in schemes such as LAC [80] were found to be unsuitable for generating secret polynomials. On the other hand, improvements in the NTT polynomial multiplications, such as using K-reduction algorithm [79], or just-in-time generation of module lattices, almost became the standard choice. Therefore, the NIST standardization process does not mark a zenith in the research and development of lattice-based or PQC; rather, it has established a framework and set a course for future advancement in PQC.

In this work, we have decided to evaluate various design choices for constructing PQ KEM. We have particularly chosen the hard lattice problem learning with rounding (LWR). LWR is a relatively less used hard problem when designing lattice-based cryptographic schemes. The LWR problem is a de-randomized variant of the LWE problem where a deterministic rounding to a smaller modulus replaces the error sampling. This problem was introduced by Banerjee et al. [13] in 2012. Several works have been done on the hardness of the LWR problem and deduced that the LWR problem is as hard as the LWE problem [8, 9, 25]. Nevertheless, in the context of PQ KEM Saber [14], one of the finalists in the NIST procedure, we have seen quite some intriguing results on the LWR-based schemes. In particular, the major reasons that motivated us to explore the LWR-based PQ KEMs further are described below.

LWR-based schemes require fewer pseudo-random numbers than LWE-based schemes, as errors are not required to be sampled explicitly here. The error is generated inherently from rounding operations, which helps to gain better performance. The rounding modulus is smaller than the modulus of the LWE problem. Therefore for similar security levels, it results in smaller public-key sizes and ciphertext sizes. This also implies lesser bandwidth compared to the schemes based on the LWE problem. Although Kyber is based on the module-LWE (MLWE) problem, it also uses rounding on the encapsulation procedure (Compress function) to reduce the ciphertext size. In terms of performance, module-LWR (MLWR) based scheme Saber outperforms MLWE-based scheme Kyber in the cortex-M4 platform when an NTT-based multiplier is used for Saber (shown in Table 6). As NTT-based polynomial multiplication takes a similar amount of cycles for Saber and Kyber, Saber doesn't require expensive error sampling. Also, Saber's auxiliary functions, such as compress, decompress, encode, and decode operations, are simple and cheaper than Kyber's, thanks to its power-of-two moduli.

The choice of power-of-two moduli helps Saber achieve efficient hardware implementation as well. LWR-based schemes, in general, use Toom-Cook based polynomial multiplication instead of NTT-based polynomial multiplication. It helps to reduce the area requirements to implement LWR-based schemes in hardware compared to the LWE-based schemes. To perform NTT multiplication efficiently, the twiddle factors need to be stored in the memory. Also, the secret polynomial is smaller in the LWR-based scheme than LWE-based scheme, because in LWE-based scheme NTT needs to be performed on the secret polynomial, and it increases the memory requirement to store the secret key after the NTT. LWE-based schemes need to use a prime reduction algorithm (for Kyber, it is Montgomery and Barrett reduction), which also costs memory. However, no fancy reduction algorithm is required for LWR-based schemes due to its power-of-two moduli. All the factor mentioned above helps LWR-based schemes to a resource-scarce cryptographic application. There is already an implementation of MLWR-based KEM Saber in the application-specific integrated circuit (ASIC) available that uses the lowest area, lowest power, and low energy [48, 50].

There exist several physical attack *i.e.* side-channel attacks (SCA) and fault injection attacks(FIA) [10, 54, 63, 74, 84, 85, 94, 94] on both lattice-based signatures and KEMs. Since in this work, we are mostly concerned with KEMs, we will keep our discussion regarding physical attacks on PQC limited to KEMs only. Masking [34] is a well-known and provably secure countermeasure against SCA. The integration of the masking technique into a KEM scheme incorporates huge performance overhead. However, the performance overhead of the LWR-based KEM Saber after masking is comparatively less than the LWE-based scheme Kyber. State-of-the-art first-order masked Saber performs slightly better (4%) than Kyber [17, 56], but the performance difference between Saber and Kyber is considerably much for higher-order masking. State-of-the-art second-order and third-order masked Saber perform 53% and 48% better than Kyber, respectively [30, 75]. There are several components, such as arithmetic-to-Boolean conversion, Boolean-to-arithmetic conversion, compress, encode, and decode, that are cheaper when power-of-two modulus (used in LWR-based schemes) is used instead of prime modulus (used in LWE-based schemes). The secret and error of the

LWE instances in the LWE-based schemes are generated from the same seed. There is a fault attack on LWE-based KEM where the successfully injected fault in the seed results in an LWE instance where the error and secret are the same [97]. It breaks the hardness assumption of the LWE problem. However, as the LWR problem has no explicit error sampling, LWR-based schemes are naturally protected against this kind of fault attack.

Although NIST has selected Kyber as their first post-quantum secure KEM standard, several other standardization efforts are still in process, such as the Korean post-quantum competition (KPQC) [73]. It is currently in its second round. Smaug is based on a combination of MLWE and MLWR problems, and its design is inspired by the initial Scabbard paper [18]. Smaug [35] is one of the second-round KEM candidates. The currently selected PQC algorithms have been designed to address a variety of problems for many different applications. In particular, they have not been designed specifically for resource-constrained or Internet of Things (IoT) devices. Due to the rapid proliferation of these devices in almost every part of our digital ecosystem, they have become ubiquitous. However, due to their small sizes, it is often difficult to equip them with strong security measures. Due to these reasons often they become the weakest part of any security protocol. Therefore, there is an urgent need to design PQC schemes specifically for these devices.

There are two ways to design lightweight schemes for resource-constrained devices: design new schemes (different design components, different parameters, etc.) from scratch for resource-constrained devices or implement the existing schemes in a lightweight manner by probably trading off efficiency or reducing the security. The work on designing lightweight PQC has just begun to gain attention [33, 45]. Recently, a lightweight MLWE-based KEM, Rudraksh, has been proposed for resource-constrained devices [76]. Therefore, we believe that our current work on studying the exploration of various design and parameter choices will have a valuable positive impact on the seamless transition from classical to PQC. We also think that this study will help construct efficient schemes and improve state-of-the-art practices. In fact, NIST historically includes and updates their already standardized cryptographic primitives with efficient ones. For example, NIST first standardized the elliptic curves digital signature algorithm in 1999 [88] and recommended 15 elliptic curves. After that, throughout the 2.5 decades, NIST has been modifying its recommended list of elliptic curves [86, 87], and the last modification was performed in 2023 [89]. Therefore, the progress achieved in this work will benefit to improve the state-of-the-art of post-quantum cryptography.

Contribution: We propose Scabbard, a suite with three new LWR-based key-encapsulation mechanisms (KEMs): Florete, Espada, and Sable. To implement these three schemes efficiently, we utilized one of the NIST's finalist KEMs, Saber's optimized software and hardware implementations, and modified it according to our scheme requirements. In this paper, we extend our earlier work, Scabbard's initial suite [18] by proposing parameters for several new security versions of previously proposed schemes and also implementing them in software. We also present unified hardware implementations of a (medium) security version of all these three schemes. Below, we briefly elaborate on all of our contributions.

- Florete is the first candidate of the Scabbard suite, and it is based on the ring-LWR (RLWR) hard problem. This KEM is designed to provide performance efficiency over the other LWE/LWR-based schemes. We proposed only the medium (NIST-3) security version in the initial paper. As an extension, we propose a low (NIST-1) and a high (NIST-5) security version of the Florete scheme in this paper. We show that all three security versions of Florete maintain the initial design rationale, and Florete performs better than most of the other lattice-based KEMs on the software platforms.
- Espada is the second scheme of this suite, and its hardness depends on the MLWR problem. However, the size of the polynomial used in this scheme is as small as 64, which is the first of its kind. The small polynomial size

makes this scheme suitable for resource-constraint devices and also highly parallelizable in hardware. In this paper, we propose a low and high-security version of the Espada. Together with the previous medium security version of Espada [18], this scheme now has three security versions, which broadens its applicability. We also show that all security versions of this scheme use less stack memory than most of the other lattice-based KEMs on the software platforms.

- We explored parameter sets similar to Saber's and obtained slightly reduced parameter sets that provide similar security. This new variant of Saber is the third scheme of our suite, Sable. It is based on the MLWR problem, and the polynomial used in this scheme is of the same size as Saber (256). This scheme can also be considered an efficient variant of Saber. We implemented all three security versions of Sable and show that it performs better than Saber and requires less stack memory on software platforms. We also show that Sable performs better than Saber and has less memory footprint when implemented on hardware platforms.
- We provide efficient implementations of all the schemes of Scabbard (also for all the security variants) on Intel's general-purpose processor and further optimize them with advanced vector instructions (AVX2). We also provide efficient implementations of Scabbard's schemes on the ARM Cortex-M4 platform. Low and high-security versions of Florete and Espada are implemented efficiently on all these software platforms for this paper. We compare our schemes with state-of-the-art schemes, such as the NIST standard Kyber, the Federal Office for Information Security's (BSI) standard Frodo, and several KPQC schemes on software platforms. We show that Florete performs better and Espada uses less amount of stack memory than most of the state-of-the-art KEMs on the Cortex-M4 platforms for all the security versions.
- MLWR-based scheme Saber is implemented using Toom-Cook (TC) based polynomial multiplication, but Chung et al. [36] improved its performance with a number-theoretic transformation (NTT) based polynomial multiplication and then Abdulrahman et al. [1] improved it even more. To show that this result can be extended for the schemes of our suite, we propose an implementation of Sable with number theoretic transformation (NTT) based polynomial multiplication on the Cortex-M4 platform and call it SableNTT. Our SableNTT not only performs better than SaberNTT (Saber with NTT-based polynomial multiplication) but also performs better than Kyber-Speed (Kyber's implementation optimized for speed).
- We implement Florete, Espada, and Sable as full instruction-set coprocessor architectures on hardware (medium security version, NIST-3). By integrating and optimizing all building blocks, our design can compute all KEM operations in hardware: key generation, encapsulation, and decapsulation. As most individual components use non-multiples of 8-bit operands, hardware implementations become increasingly complex. We discuss our approach and optimized design, leading to reduced cycle counts and area counts. We utilize the polynomial multiplier architectures proposed in the initial paper. We show that our Sable implementation outperforms Saber, and all of the Scabbard schemes have comparable performance with state-of-the-art KEMs on hardware platforms.

2 PRELIMINARIES

2.1 Notation

We represent the set of integers modulo q by $\mathbb{Z}_q$ for a positive integer q. We use $\mathcal{R}_q^n$ to denote the quotient ring $\mathbb{Z}_q[x]/(x^n + 1)$ or $\mathbb{Z}_q[x]/(x^n - x^{n/2} + 1)$. The ring with l length vectors over $\mathcal{R}_q^n$ is denoted by $(\mathcal{R}_q^n)^l$, and the ring of $m \times l$ matrices over $\mathcal{R}_q^n$ is refereed by $(\mathcal{R}_q^n)^{m\times l}$. We denote single polynomials by lower case letters and matrices by upper case letters. We denote by $\{x_i\}_{0\leq i\leq t}$ to the set of $t+1$ elements $\{x_0, x_1, \ldots, x_t\}$ from the same ring $\mathcal{R}$. If x is sampled from the set S according to the distribution χ, then we use $x \leftarrow \chi(S)$. When x is generated from a seed $\mathtt{seed}_x$ using some pseudo-random number generator according to the distribution χ over the set S, then we denote it by $x \leftarrow \chi(S;\ \mathtt{seed}_x)$. We denote the uniform distribution by $\mathcal{U}$. The centered binomial distribution (CBD) with standard deviation $\sqrt{\mu/2}$ is refereed as β_μ. We use $\cdot$ to indicate matrix-vector and vector-vector multiplications. Here, we use scaling down function $\lfloor\cdot\rceil_p : \mathbb{Z}_q \longrightarrow \mathbb{Z}_p$ defined by $\lfloor x\rceil_p = \lfloor(q/p)x\rceil$, where $q > p$ and the rounding function $\lfloor y\rceil$ outputs the closest integer to the real number y, and during ties rounded upwards e.g. $\lfloor 1/2\rceil = 1$. The operations $\lfloor\cdot\rceil_p$ can be extended for matrices and vectors by applying them coefficient-wise. Throughout this paper, the multiplication of two n degree polynomials over the ring $\mathcal{R}_q^n$ is mentioned as $n \times n$ polynomial multiplication. We use $\cdot$ to represent multiplication between two polynomials, two vectors of polynomials, or one matrix and one vector of polynomials, depending on the context.

2.2 Learning with Rounding Problem

The decision version of LWE problem [98] states that given $\mathbf{A} \leftarrow \mathcal{U}(\mathbb{Z}_q^{m\times l})$ and $\mathbf{s}$ and $\mathbf{e}$ are sampled according to following respective small distributions $\beta_\mu(\mathbb{Z}_q^l)$ and $\beta_\mu(\mathbb{Z}_q^m)$, distinguishing between the LWE sample $(\mathbf{A}, \mathbf{b} = \mathbf{A}\cdot\mathbf{s}+\mathbf{e}) \in \mathbb{Z}_q^{m\times l} \times \mathbb{Z}_q^m$ and $(\mathbf{A}, \mathbf{b}') \in \mathbb{Z}_q^{m\times l} \times \mathbb{Z}_q^m$ is hard, when $\mathbf{b}'$ is sampled uniformly from $\mathbb{Z}_q^m$. The LWR problem [8, 13] is a variation of the LWE problem, and the LWR sample is constructed as $(\mathbf{A}, \mathbf{b} = \lfloor\mathbf{A}\cdot\mathbf{s}\rceil_p = \lfloor(q/p)\mathbf{A}\cdot\mathbf{s}\rceil) \in \mathbb{Z}_q^{m\times l} \times \mathbb{Z}_p^m$, where $\mathbf{s} \leftarrow \beta_\mu(\mathbb{Z}_q^l)$. Here, we do not need an explicit sampling of $\mathbf{e}$ rather, it generates implicitly from rounding. The decision version of LWR problem states that given $\mathbf{A} \in \mathbb{Z}_q^{m\times l}$, it is hard to differentiate between the LWR sample $(\mathbf{A}, \mathbf{b} = \lfloor\mathbf{A}\cdot\mathbf{s}\rceil_p) \in \mathbb{Z}_q^{m\times l} \times \mathbb{Z}_p^m$ and $(\mathbf{A}, \mathbf{b}') \in \mathbb{Z}_q^{m\times l} \times \mathbb{Z}_p^m$, where $\mathbf{s} \leftarrow \beta_\mu(\mathbb{Z}_q^l)$ and $\mathbf{b}' \leftarrow \mathcal{U}(\mathbb{Z}_p^m)$.

Ring-LWE (RLWE) problem is a variant of the LWE problem based on structure lattice and is proposed in [81] to improve the practicality and efficiency of cryptographic schemes. In the RLWE, $\mathbf{A}$, $\mathbf{s}$, $\mathbf{e}$, and $\mathbf{b}$ of the LWE are all replaced by polynomials of the ring $(\mathcal{R}_q^n)$. Similar to the RLWE, we can define the decision version of the RLWR problem, which states that the RLWR sample $(\mathbf{a}, \mathbf{b} = \lfloor\mathbf{a}\cdot\mathbf{s}\rceil_p) \in \mathcal{R}_q^n\times\mathcal{R}_p^n$ and $(\mathbf{a}, \mathbf{b}') \in \mathcal{R}_q^n\times\mathcal{R}_p^n$ are hard to distinguish, where $\mathbf{s} \leftarrow \beta_\mu(\mathcal{R}_q^n)$ and $\mathbf{b}' \leftarrow \mathcal{U}(\mathcal{R}_p^n)$. The ring version offers better efficiency and practicality compared to the cryptosystem based on standard lattices of similar security. However, due to the presence of additional structures, many researchers are skeptical about their hardness. Therefore, as a trade-off between security and efficiency, the MLWR problem was introduced [78], which states that it is hard to differentiate between the MLWR sample $(\mathbf{A}, \mathbf{b} = \lfloor\mathbf{A}\cdot\mathbf{s}\rceil_p) \in (\mathcal{R}_q^n)^{l\times l} \times (\mathcal{R}_p^n)^l$ and $(\mathbf{A}, \mathbf{b}') \in (\mathcal{R}_q^n)^{l\times l} \times (\mathcal{R}_p^n)^l$, where $\mathbf{s} \leftarrow \beta_\mu((\mathcal{R}_q^n)^l)$ and $\mathbf{b}' \leftarrow \mathcal{U}((\mathcal{R}_p^n)^l)$. The rank of the underlying lattice of this MLWR problem is $n \times l = n'$. MLWR has less structure than RLWR, and the expensive matrix-vector multiplication of the standard LWR problem is replaced by efficient polynomial multiplication in the MLWR problem.

The module lattice-based problem can be used as generic construction, as all the MLWR problems with $n = n'$ and $l = 1$ are classified as RLWR problems, and all the MLWR problems with $n = 1$ and $l = n'$ are categorized as standard LWR problems. At this moment, there is no attack that provides any advantage to the adversary for MLWR or RLWR problems over the standard LWR problem. Therefore, if the rank of the underlying lattice problem is the same, then the security provided by the problem is the same. Henceforth, we will use the MLWR problem to denote different variations of the LWR problem.

2.3 Construction of Generic LWR-based KEM

The LWR-based public-key encryption (PKE) scheme is used to construct an LWR-based key-encapsulation mechanism (KEM), and we illustrate the PKE scheme in Fig. 1. It consists of three algorithms (i) key-generation (`LWR.PKE.KeyGen`),

(ii) encryption (LWR.PKE.Enc), and (iii) decryption (LWR.PKE.Dec). Firstly, the LWR.PKE.KeyGen algorithm generates the public key and secret key pair. Secondly, the LWR.PKE.Enc algorithm uses the public key to encrypt the message m and to produce ciphertext. Lastly, the LWR.PKE.Dec algorithm decrypts the received ciphertext to the message m'.

Three quotient rings $\mathcal{R}_q^n, \mathcal{R}_p^n, \mathcal{R}_t^n$ has been used here, and $t < p < q$. The constants $\epsilon_q = \log_2(q), \epsilon_p = \log_2(p), \epsilon_t = \log_2(t)$ are used to construct the constant polynomials h_2, h_3, and by the vector of constant polynomials $\mathbf{h_1}$. Each coefficient of the constant polynomials h_2 and h_3 are $2^{(\epsilon_q-\epsilon_p-1)}$ and $(2^{(\epsilon_p-B-1)} - 2^{(\epsilon_p-\epsilon_t-1)})$, respectively. The value of each coefficient of the vector with constant polynomials $\mathbf{h_1}$ is $2^{(\epsilon_q-\epsilon_p-1)}$. The extendable-output function $\mathsf{XOF} : \{0,\ 1\}^{256} \longrightarrow \{0,\ 1\}^*$ is a pseudorandom number generator realized with SHAKE-128.

As the LWR-based schemes are not perfect, there is always a possibility of a decryption failure, i.e. the encrypted message m and decrypted message m' are not equal even when the scheme is executed properly. The decryption failure probability depends on the decryption noise $d = v'' - v'$. The decryption failure will not occur if the decryption noise d satisfies the following relation $|d| \leq \frac{p}{2^{B+1}}(1 - \frac{1}{t})$ [29]. Therefore, if $t = 2_t^{\epsilon}$ is large enough, then the decryption failure probability, δ, becomes negligible. Eventually, it makes the corresponding LWR-based PKE scheme $(1 - \delta)$ correct.

```
LWR.PKE.KeyGen()
seed_A ← $\mathcal{U}(\{0,\ 1\}^{256})$                                        ▷ seed of the public matrix $\boldsymbol{A}$
$\boldsymbol{A} \leftarrow \mathcal{U}((\mathcal{R}_q^n)^{l\times l};\ \mathsf{seed}_{\boldsymbol{A}})$                  ▷ $\boldsymbol{A}$ is generated using XOF function over $\mathsf{seed}_{\boldsymbol{A}}$
$\mathsf{seed}_{\boldsymbol{s}} \leftarrow \mathcal{U}(\{0,\ 1\}^{256})$                                        ▷ seed of the secret vector $\boldsymbol{s}$
$\boldsymbol{s} \leftarrow \beta_\mu((\mathcal{R}_q^n)^l;\ \mathsf{seed}_{\boldsymbol{s}})$                           ▷ $\boldsymbol{s}$ is generated using CBD $\beta_\mu$ over the $\mathsf{XOF}(\mathsf{seed}_{\boldsymbol{s}})$
$\boldsymbol{b} \leftarrow ((\boldsymbol{A}^T \cdot \boldsymbol{s} + \boldsymbol{h_1}) \bmod q) \gg (\epsilon_q - \epsilon_p) \in (\mathcal{R}_p^n)^l$   ▷ Performing rounding operation on $\boldsymbol{A}^T \cdot \boldsymbol{s}$ to create $\boldsymbol{b}$
return $(pk = (\mathsf{seed}_{\boldsymbol{A}},\ \boldsymbol{b}),\ sk = (\boldsymbol{s}))$                            ▷ $pk$ public key and $sk$ secret key

LWR.PKE.Enc($pk = (\mathsf{seed}_{\boldsymbol{A}}, \boldsymbol{b}), m \in R_2; r$)                          ▷ $pk$ is sent via insecure channel
$\boldsymbol{A} \leftarrow \mathcal{U}((\mathcal{R}_q^n)^{l\times l};\ \mathsf{seed}_{\boldsymbol{A}})$         ▷ $\boldsymbol{A}$ is re-generated using XOF function over public $\mathsf{seed}_{\boldsymbol{A}}$
if: $r$ is not specified:
    $r \leftarrow \mathcal{U}(\{0,\ 1\}^{256})$                                              ▷ seed of the $\boldsymbol{s'}$
$\boldsymbol{s'} \leftarrow \beta_\mu((\mathcal{R}_q^n)^l;\ r)$                              ▷ $\boldsymbol{s'}$ is produced using CBD $\beta_\mu$ over the $\mathsf{XOF}(\mathsf{seed}_{\boldsymbol{s'}})$
$\boldsymbol{u} \leftarrow ((\boldsymbol{A} \cdot \boldsymbol{s'} + \boldsymbol{h_1}) \bmod q) \gg (\epsilon_q - \epsilon_p) \in (\mathcal{R}_p^n)^l$         ▷ Creates $\boldsymbol{b'}$ key contained part of the ciphertext
$v' \leftarrow \boldsymbol{b}^T \cdot (\boldsymbol{s'} \bmod p) + h_2 \in \mathcal{R}_p^n$
$v \leftarrow (v' - 2^{\epsilon_p-B} m \bmod p) \gg (\epsilon_p - \epsilon_t - B) \in \mathcal{R}_{2^B t}^n$            ▷ $\boldsymbol{v}$ message contained part of the ciphertext
return $c = (\boldsymbol{u},\ v)$                                                   ▷ Ciphertext c

LWR.PKE.Dec($sk = \boldsymbol{s}, c = (\boldsymbol{u}, v)$)                         ▷ $sk$ is stored and $c$ is delivered via insecure channel
$v'' \leftarrow \boldsymbol{u}^T \cdot (\boldsymbol{s} \bmod p) + h_2 \in \mathcal{R}_p^n$
$m' \leftarrow (v'' - 2^{\epsilon_p-\epsilon_t-B} v + h_3) \bmod p \gg (\epsilon_p - B) \in \mathcal{R}_{2^B t}^n$ ▷ Recover message by decoding the ciphertext $c$
return $m'$
```

Fig. 1. Generic LWR.PKE

The aforementioned LWR-based PKE scheme is indistinguishable against chosen plaintext attacks (IND-CPA) and can be converted to an indistinguishable under adaptive chosen ciphertext attacks (IND-CCA) KEM with a modified version of Fujisaki-Okamoto transformation [47] proposed by Hofheinz *et al.* [60]. Authors show that if the underlying PKE scheme is $(1 - \delta)$ correct, then the KEM is also $(1 - \delta)$ correct. The KEM is S bit post-quantum secure only when the failure probability $\delta < 2^{-S}$ [64].

```
LWR.KEM.KeyGen()
$(pk\,sk) = ((seed_{\boldsymbol{A}},\ \boldsymbol{b}),\ \boldsymbol{s}) \leftarrow$ LWR.PKE.KeyGen() ▷ The key-generation of the PKE is called to generate $(pk\,sk)$
$pkh \leftarrow \mathcal{H}(pk)$ ▷ Hash of $pk$ is part of the secret key
$z \leftarrow \mathcal{U}(\{0,\ 1\}^{256})$ ▷ random $z$ is used by decapsulation in cases of decryption failure
return $(\overline{pk} = pk = (seed_{\boldsymbol{A}},\ \boldsymbol{b}),\ \overline{sk} = (sk,\ z,\ pkh) = (\boldsymbol{s},\ z,\ pkh))$ ▷ public key $\overline{pk}$, secret key $\overline{sk}$

LWR.KEM.Encaps($\overline{pk} = (seed_{\boldsymbol{A}},\ \boldsymbol{b})$)
$m' \leftarrow \mathcal{U}(\{0,\ 1\}^{256})$ ▷ message sampled
$m \leftarrow$ arrange_msg($m'$) ▷ message arranged for sending as input in LWR.PKE.Enc
$(\hat{K}, r) \leftarrow \mathcal{G}(\mathcal{H}(pk), m)$ ▷ $r$ is used as seed to generate $s'$ in LWR.PKE.Enc during PKE encryption
$c \leftarrow$ LWR.PKE.Enc($pk, m; r$) ▷ The encryption of the PKE is called to generate ciphertext $c$
$K \leftarrow$ KDF($\hat{K}, \mathcal{H}(c)$) ▷ $\hat{k}$ and ciphertext $c$ are used to the generate shared key $K$
return $(c,\ K)$ ▷ ciphertext of the KEM is $(c,\ K)$

LWR.KEM.Decaps($\overline{sk} = (\boldsymbol{s},\ z,\ pkh), pk = (seed_{\boldsymbol{A}},\ \boldsymbol{b}),\ c$)
$m'' \leftarrow$ LWR.PKE.Dec($\boldsymbol{s}, c$) ▷ The decryption of the PKE is called to decrypt $c$
$m' \leftarrow$ original_msg($m''$) ▷ Inverse of the arrange_msg to get back the original message
$(\hat{K}',\ r') \leftarrow \mathcal{G}(pkh, m')$ ▷ $r'$ is used to generate $s'$ in re-encryption and $\hat{K}'$ is used for shared key generation
$c_* \leftarrow$ LWR.PKE.Enc($pk, m'; r'$) ▷ re-encryption with decrypted message $m'$
if: $c = c_*$ ▷ equality check of the public ciphertext $c$ and re-encrypted ciphertext $c_*$
    $K \leftarrow$ KDF($\hat{K}', \mathcal{H}(c)$) ▷ the equality check satisfies and $K$ is the valid shared key
else:
    $K \leftarrow$ KDF($z, \mathcal{H}(c)$) ▷ the equality check fails and $K$ is an invalid shared key
return $(K)$
```

Fig. 2. Generic LWR.KEM

The CCA-secure LWR-based KEM based on the CPA-secure LWR-based PKE is presented in Fig. 2. This KEM consists of three algorithms: (i) key-generation (LWR.KEM.KeyGen), (ii) encapsulation (LWR.KEM.Encaps), and (iii) decapsulation (LWR.KEM.Decaps). Here, the key-generation algorithm generates the public key and secret key pair $(\overline{pk},\ \overline{sk})$. Secondly, the encapsulation algorithm uses the public key $\overline{pk}$ to encrypt the message and to produce ciphertext c and the session key K. Lastly, in the decapsulation algorithm, we decrypt the received ciphertext to the message then we re-encrypt the decrypted message using the public key. If the re-encrypted ciphertext is equal to the received ciphertext, then the algorithm outputs the session key K; else outputs a random key. In these algorithms, we use two hash functions $H : \{0,\ 1\}^* \longrightarrow \{0,\ 1\}^{256}$ realized by SHA3-256 and $\mathcal{G} : \{0,\ 1\}^* \longrightarrow \{0,\ 1\}^{512}$ implemented with SHA3-512. In LWR.KEM.Encaps and LWR.KEM.Decaps, the arrange_msg : $\{0,\ 1\}^{256} \longrightarrow \mathcal{R}_q^n$ function is used, which converts the 256 bits message to the message polynomial in $\mathcal{R}_q^n$. The inverse of arrange_msg function original_msg : $\mathcal{R}_q^n \longrightarrow \{0,\ 1\}^{256}$ is required in LWR.KEM.Decaps. It converts the message polynomial in $\mathcal{R}_q^n$ to the 256 bits message.

3 SCABBARD SUITE OF LWR-BASED KEMS

We present the schemes of the suite Scabbard in this section. These schemes have been designed to improve the state-of-the-art of the efficient lattice-based KEM. This suite consists of three different designs of LWR-based KEMs (i) Florete, (ii) Espada, and (iii) Sable. All three schemes follow the generic LWR-based KEM construction. Florete is designed to achieve better performance. Espada is designed to use less memory footprint when implemented for resource-constrained devices and can also be implemented efficiently in hardware by using its high parallelism. Sable is designed to provide a trade-off between performance and memory usage. As we can see from Fig. 1, for a LWR-based KEM, everything is the same except the choice of the ring/ module parameters n and l, the CBD parameter μ, moduli $q,\ p,\ t$. Therefore, the polynomial multiplication, message encoding and decoding, and the secret sampler used in these three schemes are different. We will discuss these different aspects of the KEMs and describe their design rationale in the following sections. In Scabbard's KEMs design, one of the important aspects is we tried to maximally utilize already developed optimized software and hardware modules of LWR-based schemes.

3.1 Florete: RLWR based KEM

This scheme is based on the RLWR hard problem, and therefore the ring/modulus parameter l is equal to 1. The public matrix $\mathbf{A}$, secret vector $\mathbf{s}$, and public key vector $\mathbf{b}$ are all polynomials and are elements of the ring $\mathcal{R}_q^n$. The parameter n and the ring $\mathcal{R}_q^n$ vary for different security versions of Florete. We choose $n = 512$ for the low-security version, $n = 768$ for the medium-security version, and $n = 1024$ for the high-security version. We are required to use an irreducible polynomial to construct the ring R_q^n for the RLWR problem [81] (otherwise hardness of the RLWR problem reduces). Generally, $x^n + 1$ is chosen as an irreducible polynomial to construct the ring R_q^n, but $x^{768} + 1$ is not an irreducible polynomial. Therefore, the irreducible polynomial $(x^{768} - x^{384} + 1)$ is applied to construct R_q^n for $n = 768$.

Polynomial Multiplication

Polynomial multiplication is one of the fundamental operations performed during all three algorithms of a KEM, and it is one of the most time-consuming operations. The procedure of this multiplication depends on the two parameters of $\mathcal{R}_q^n$, which are n and modulus q. As mentioned earlier, we plan to utilize the optimized software and hardware modules developed for LWR-based schemes (e.g., Saber) during the NIST competition for easier adaptation. Polynomial multiplication is one of the modules whose implementation has been optimized in several works [16, 65, 82, 100]. LWR-based schemes can not use fast number theoretic transformation for polynomial multiplication because the modulus q and n are not co-prime ($\mathtt{gcd}(q, n) > 1$). The next best option for the $n \times n$ polynomial multiplication is utilizing Toom-Cook or Karatsuba multiplication, which has been used and optimized for the MLWR-based KEM Saber [82, 100]. Therefore, we have decided to re-purpose Saber's efficient 256×256 multiplier for Florete's $n \times n$ multiplication. We use Saber's efficient 256×256 multiplier for implementing all three polynomial multiplications of Florete. The 256×256 polynomial multiplication of Saber is implemented by using a layer of Toom-Cook4 multiplication followed by two layers of Karatsuba multiplication, and the last stage is 16×16 schoolbook multiplication.

There is a small problem in using Saber's multiplier in Florete. We target to fit a coefficient of the multiplier polynomial fit into 16 bit space for efficient implementation in vector processors, small microcontrollers (e.g. Cortex-M4), etc. Even though the coefficients of the multiplier are less than or equal to 16 bit, we need to save some extra space when performing division by some g, which is a divisor of 2. The reason is $\mathtt{gcd}(g,\ q) \geq 2$, and the inverse of g does not exist in $\mathbb{Z}_q$. In this case, let us assume y/g needs to be computed. If $g = h * 2^w$, where $\mathtt{gcd}(h,\ 2) = 1$, then $\mathtt{gcd}(h,\ q) = 1$ (as q is a power-of-2 modulus). We first compute h^{-1}, and then we multiply it with g and perform w right shift afterward. Therefore we need to store w extra bits while performing $g \cdot h^{-1}$. There are several such divisions by g, where $g = h * 2^w$ are needed while using Toom-Cook multiplications. The maximum value of such w is equal to 1 for Toom-Cook 3-way multiplication, whereas w is equal to 3 for Toom-Cook 4-way multiplication. For the 512×512 polynomial multiplication, we use one extra layer of Karatsuba multiplication on top of Saber's 256×256 multiplication. Therefore, the $\log_2$ of the modulus q, ϵ_q need to be ≤ 13 for the low-security version of Florete. For the 768×768 polynomial multiplication, we add an extra layer of Toom-Cook 3-way multiplication on top of the 256×256 multiplication. Here, the ϵ_q need to be

≤ 12 $(= 16 - 3 - 1)$ (It is not possible with Saber's modulus q, which is 2^{13}). We apply Toom-Cook 4-way multiplication on top of the 256×256 multiplication for the 1024×1024 polynomial multiplication. The modulus ϵ_q need to be ≤ 10 $(= 16 - 3 - 3)$ in this case.

Now, we will compare the number of 256×256 polynomial multiplications used in Saber with Florete. As Saber is based on the MLWR problem, it has a module structure and the parameter $l > 1$. Therefore several 256×256 polynomial multiplications are needed for matrix-vector multiplications (e.g. $\boldsymbol{A} \cdot \boldsymbol{s}$) and vector-vector multiplications (e.g. $\boldsymbol{b}^T \cdot \boldsymbol{s'}$), which are used in all three (key-generation, encapsulation, and decapsulation) algorithms of all three security versions of Saber. These exact numbers are provided in Table 1. For example, the key-generation, encapsulation, and decapsulation algorithm of the medium-security version of Saber requires 9, 12, and 15 polynomial multiplication (256×256), respectively.

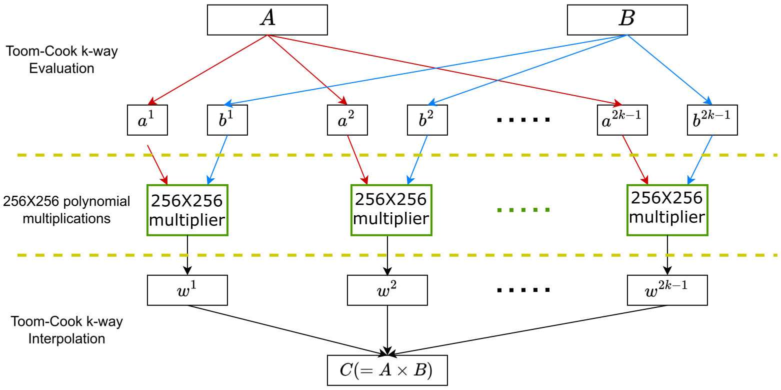

Fig. 3. Polynomial multiplication used in Florete. The values of k for low, medium, and security versions of Florete are 2, 3, and 4, respectively.

Florete is an RLWR-based scheme. So, all matrix-vector multiplications (e.g. $\boldsymbol{A} \cdot \boldsymbol{s}$) and vector-vector multiplications (e.g. $\boldsymbol{b}^T \cdot \boldsymbol{s'}$) are just a single polynomial multiplication in Florete. The key-generation, encapsulation, and decapsulation algorithms of Florete require 1, 2, and 3 $n \times n$ polynomial multiplications, respectively. As mentioned earlier, we apply an extra layer of Karatsuba multiplication on top of Saber's 256×256 multiplication for 512×512 polynomial multiplication of the low-security version of Florete. Here, we perform 3 256×256 polynomial multiplications for a 512×512 polynomial multiplication (as displayed in Fig. 3). There are some other steps which are interpolation and reduction, to complete the whole multiplication. However, the performance cost of these steps is negligible compared to the total number of polynomial multiplications. Therefore, the low-security version of Florete needs 3, 6, and 9 256×256 multiplications for the key-generation, encapsulation, and decapsulation algorithms, respectively. For a 768×768 polynomial multiplication, an extra layer of Toom-Cook 3-way multiplication is added on top of 256×256 multiplication. As we are applying Toom-Cook 3-way multiplication for 768×768 polynomial multiplication, we need to perform $5 = (2 * 3 - 1)$, 256×256 polynomial multiplications (as portrayed in Fig. 3). We are applying an extra layer of Toom-Cook 4-way multiplication for 1024×1024 polynomial multiplication. So, we need to perform $7 = (2 * 4 - 1)$, 256×256 polynomial multiplications for a single 1024×1024 multiplication (as shown in Fig. 3). We provide the number of 256×256 polynomial multiplications required by all the algorithms of all the security versions of Florete in Table 1. We have included the performance of multiplications used in Florete and Saber on the Cortex-M4 platform. More detailed performance results are given in Sec. 5. We can see from Table 1 that the number of 256 × 256 multiplications used in the key-generation algorithm of the low-security version of Florete is less than the low-security version of Saber. The number of 256 × 256 multiplications used in the encapsulation algorithm of the low-security version of Florete is the same as the low-security version of Saber, whereas the decapsulation algorithm of the low-security version of Florete uses 1 more 256 × 256 polynomial multiplication than Saber. For the medium security, the number of 256 × 256 multiplications used in the key-generation and encapsulation algorithms of Florete is less than Saber. The number of 256 × 256 multiplications used in the decapsulation algorithm of the medium-security version of Florete is the same as Saber. Lastly, for the high security, the number of 256×256 multiplications used in all the algorithms of Florete is less than Saber.

Table 1. Comparison of the usage of 256 × 256 multiplications in the algorithms of Florete with Saber.

Scheme Name	Security level	#256 × 256 multiplications			Multiplication on Cortex-M4 (x1000 clock cycles)		
		KeyGen	Encaps	Decaps	KeyGen	Encaps	Decaps
Florete	Low	**3**	6	9	**121**	243	364
	Medium	**5**	**10**	15	**202**	**405**	607
	High	**7**	**14**	**21**	**300**	**600**	900
Saber	Low	4	6	8	149	223	298
	Medium	9	12	15	334	446	557
	High	16	20	24	594	743	891

Message Encoding and Decoding

Message encoding and decoding are done in the LWR-based KEM described in Fig. 2 by using the `arrange_msg` and `original_msg`, respectively. The secret payload/message (m') size is 256 bits in all the security versions of Florete, and the size of the polynomial is at least twice. So, we repeat the secret payload multiple times with the help of the $\texttt{arrange\_msg}\{0,\ 1\}^{256} \longrightarrow \{0,\ 1\}^n$ function and make its bit size the same as the size of any polynomial in the corresponding security version of Florete.

$$\texttt{arrange\_msg}(m') = \begin{cases} m'||m' & \text{if } n = 512 \\ m'||m'||m' & \text{if } n = 768 \\ m'||m'||m'||m' & \text{if } n = 1024 \end{cases}.$$

The `original_msg` function is the counter function of `arrange_msg` function and is used in the decryption algorithm. We define the `original_msg` function for each security version of Florete below. For the low security version of Florete, the $\texttt{original\_msg} : \{0,\ 1\}^{512} \longrightarrow \{0,\ 1\}^{256}$ is $\texttt{original\_msg}(m'') = m'$ and $b \in \{0, 1, \ldots, 255\}$

$$m'[b] = \begin{cases} 0 & \text{if } m''[b] + m''[b + 256] \le 0 \\ 1 & \text{else} \end{cases}.$$

For the medium security version of Florete, the $\texttt{original\_msg} : \{0,\ 1\}^{768} \longrightarrow \{0,\ 1\}^{256}$ is $\texttt{original\_msg}(m'') = m'$ and $b \in \{0, 1, \ldots, 255\}$

$$m'[b] = \begin{cases} 0 & \text{if } m''[b] + m''[b + 256] + m''[b + 512] \le 1 \\ 1 & \text{else} \end{cases}.$$

For the high security version of Florete the `original_msg` : $\{0,\ 1\}^{1024} \longrightarrow \{0,\ 1\}^{256}$ is `original_msg`$(m'') = m'$ and $b \in \{0, 1, \ldots, 255\}$

$$m'[b] = \begin{cases} 0 & \text{if } m''[b] + m''[b+256] + m''[b+512] \\ & +m''[b+768] \leq 2 \\ 1 & \text{else} \end{cases} .$$

The repetition of message bits during encoding helps to reduce the failure probability and eventually helps to achieve more security. Therefore, Florete can achieve the same level of security with a smaller modulus. Therefore, we can reduce the three modulus size $\epsilon_q < \epsilon_p < \epsilon_t$ even further than Saber (or Kyber). It reduces the requirement of pseudo-random bytes to create the public matrix **A** in Florete compared to Saber, which has been supplied in Table 2. It eventually helps to reduce the public key size of Florete compared to Kyber (the exact public key sizes are shown in Table 3). Kindly note that we have not applied any error-correction code to reduce failure probability RLWE-based scheme LAC [80], as it leads to several attacks [41, 51, 55].

Secret Distribution

The coefficient of the secret **s** (or **s′**) is sampled from centered binomial distribution, β_1. Therefore possible values of a coefficient of **s** (**s′**) are $\{-1,\ 0,\ 1\}$. It enables the possibility of very fast multiplication in the processors. In this case, multiplication can be replaced by addition and subtraction only. This method is highly advantageous to the processor, where multiplication is way more costlier than addition or subtraction (e.g. MSP430 microcontrollers). Saber's secret coefficients are from β_5, β_4, and β_3 for low, medium, and high-security versions, respectively. In comparison, the secret coefficients of Florete are from β_1 for all the security versions. It leads less pseudo-random number requirements for Florete than Saber, which has been provided in Table 2. We have shown the required clock cycles to generate the matrix **A** and the secret **s** for all the versions of Florete and Saber on the Cortex-M4 platform. More detailed performance analyses are shown in Sec. 5. The coefficients of the secret can be represented by 2 bits, which reduces the memory requirement to store the secret **s** (**s′**) in hardware compared to Saber (Saber needs 4 bits to store a coefficient of **s**). It ultimately helps Florete to have smaller secret key sizes than Saber for all the security versions (the exact numbers are shown in Table 3).

Table 2. Comparison of pseudo-random byte used in Florete with Saber.

Scheme Name	Security level	pseudo-random bytes		Performance on Cortex-M4 (x1000 clock cycles)	
		matrix **A**	secret **s** (**s′**)	matrix **A**	secret **s** (**s′**)
Florete	Low	**704**	**128**	**68**	**18**
	Medium	**960**	**192**	**83**	**32**
	High	**1280**	**256**	**110**	**34**
Saber	Low	1664	640	137	76
	Medium	3744	768	313	72
	High	6656	768	545	75

The centered binomial distribution is proposed by Alkim *et al.* [7] to replace the costly Gaussian distribution, which is hard to implement in constant-time. This distribution is used in NIST standardized Kyber and third-round finalist Saber. Therefore, we have decided to sample the secret using CBD. Here, we refrained from taking any aggressive decision for secret distribution, eg. fixing the hamming weight of the secret key like LWR-based scheme Round5 [23] or fixing the weight of the secret vector like NTRU Prime [20]. This decision has been taken to avoid any new adversarial attack due to the choice of secret distribution. The Saber team proposed a lightweight version of Saber, named uSaber [14] 2 bits secret key coefficient. More specifically, they use a uniform distribution over 2 bits numbers. There is another lattice-based scheme proposed in the ongoing Korean PQC competition [73], called Smaug [35]. The secret key of this specific scheme has each coefficient sampled from the set $\{-1,\ 0,\ 1\}$.

3.2 Espada: MLWR based KEM

The next LWR-based KEM in the Scabbard is Espada. It uses MLWR as a hard problem, and this KEM also takes advantage of module lattices like Saber and Kyber. Therefore the matrix $\mathbf{A}$ is an element of the ring $(\mathcal{R}_q^n)^{l\times l}$, and the secret vector $\mathbf{s}$ is an element of the ring $(\mathcal{R}_q^n)^l$. However, the underlying quotient ring is $\mathcal{R}_q^{64}$ ($n = 64$) constructed by the help of cyclotomic polynomial $(x^{64} + 1)$. Therefore the polynomial size of Espada is just 64, which is very small compared to the size of the polynomial in Saber or Kyber (their polynomials are of size 256). This design choice allows Espada to use less stack memory while executing in a reasonable amount of clock cycles when implemented in embedded devices, such as Cortex M4 microcontrollers. It can also be implemented in hardware with less area due to its small polynomial size. This scheme can also be implemented in hardware very fast by using multiple polynomial multiplication instances. It provides the flexibility to implement Espada using one or many polynomial multipliers depending on the application's requirements (as shown in Fig. 4), which is not possible for Saber or Kyber.

Polynomial Multiplication The ring modulus $q = 2^{15}$ for each version of Espada. Since $\epsilon_q(= \log_2(q)) = 15$ and we want to restrict each coefficient of the polynomial in 16 bits of word length (described in Sec. 3.1), we cannot use Toom-Cook 4-way multiplication for the 64×64 polynomial multiplication of Espada. So, we use a combination of Karatsuba and schoolbook multiplication for the 64×64 polynomial multiplications. During the implementation of the matrix-vector or vector-vector multiplication, we take advantage of the lazy interpolation technique [82]. However, the interpolation step in Karatsuba multiplication is smaller than the interpolation step in Toom-Cook k-way multiplication for $k > 2$. The lazy interpolation technique helps to significantly improve the performance of matrix-vector and vector-vector multiplication because the vector dimension l of Espada is large.

Saber or Kyber use 256×256 polynomial multiplication, and utilizing multiple instances of this multiplication is expensive in hardware. In fact, Mera *et al.* [83] developed Saber's 256×256 multiplier by using 7 parallel 64×64 polynomial multiplication instances together with an evaluation (TC Eval) and interpolation (TC Inter) steps of Toom-Cook 4-way algorithm (as shown in Fig. 4). [83] shows that the 64×64 schoolbook multiplication is already very fast in hardware. However, this implementation uses 28 DSP just for one 256×256 multiplication. Therefore, using multiple such multiplications will make the whole design area expensive, and then there will not be much space left for other components. In Espada, the length of the polynomial is as small as 64, so the vector dimension l needs to be quite large in order to achieve the desired security. The values of l are 10, 12, and 15, corresponding to low, medium, and high-security versions of Espada (shown in Table 3). Therefore, this scheme can also be implemented in hardware very fast by employing l (the length of the vector) parallel 64×64 polynomial multiplication instances while computing the matrix-vector (e.g.: $\mathbf{A} \cdot \mathbf{s}$) and vector-vector (e.g.: $\mathbf{b}^{\mathbf{T}} \cdot \mathbf{s'}$) multiplication. In this work, we also utilize 64×64 schoolbook multiplication as one polynomial multiplication in hardware in Espada. By design Espada has extremely parallelizable matrix-vector and vector-vector multiplication in hardware, and it aids Espada to achieve high throughput in hardware.

Message Encoding and Decoding

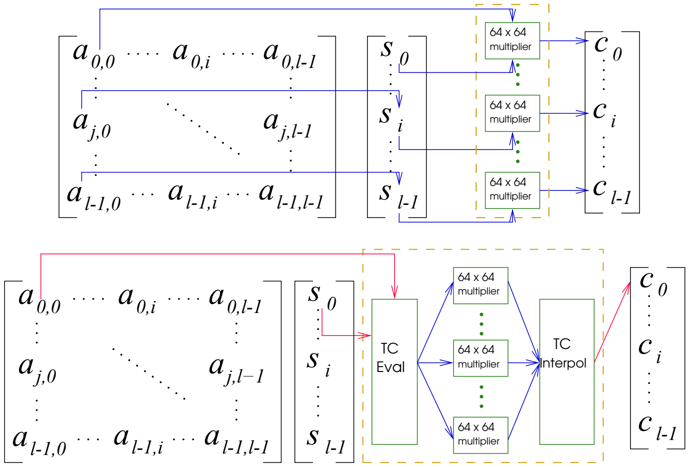

Fig. 4. Comparison between the application of parallel 64 × 64 polynomial multiplication in Espada (top) and Saber (bottom). The blue line represents parallel execution, and the red line denotes serial execution.

The degree of the polynomial for all the versions of Espada is 64, and the secret payload (m') size is 256 bits. So, we use one coefficient to hide multiple (4) bits of secret payloads and $B = 4$. Here, the function $\mathtt{arrange\_msg} : \{0,\ 1\}^{256} \longrightarrow \{0,\ 1,\ 2,\ \ldots,\ 15\}^{64}$ is $\mathtt{arrange\_msg}(m') = m$ and $b \in \{0,\ 1,\ \ldots,\ 64\}$, then $m[b] = m'[4*b]||m'[4*b+1]||m'[4*b+2]||m'[4*b+3]$. The function $\mathtt{original\_msg} : \{0, 1, 2, \ldots, 15\}^{64} \longrightarrow \{0,\ 1\}^{256}$ is $\mathtt{original\_msg}(m'') = m'$ and $b \in \{0, 1, \ldots, 255\}$, then $m'[b] = (m''[b_1] \gg b_2)\&1$, where $b = 4 * b_1 + b_2$.

Secret Distribution

In Espada, the coefficient of the secret $\mathbf{s}$ ($\mathbf{s}'$) is sampled according to the centered binomial distribution, β_3, for all the security versions. So, each secret coefficient is from the set $\{-3,\ -2,\ \cdots, 3\}$. These secret coefficients can be represented by 3 bits, but the unpacking becomes fairly costly in that case. Therefore, the cost-effective way to store the secret $\mathbf{s}$ is to reserve 4 bits for each coefficient.

3.3 Sable: an Alternate Saber

Sable can be viewed as an improved version of Saber. Like Saber, Sable uses the MLWR structure, and the underlying quotient ring of Sable is the same as Saber ($x^{256} + 1$). In this scheme, we have readjusted the parameters of Saber. The modulus size of the quotient ring q for Sable is 2^{11}, and it is smaller than Saber's modulus $q = 2^{13}$ (shown in Table 3). The public-key modulus p of Sable (2^9) is also smaller than Saber (2^{10}) for the low and medium security version, which assists in Sable having shorter public keys (given in Table 3).

Polynomial Multiplication

Sable can utilize the same 256×256 polynomial multiplication, as the modulus is less than 13 bits. Also, the polynomial multiplication of Sable is less costly than Saber in hardware, thanks to its smaller modulus.

Message Encoding and Decoding

The degree of the polynomial for all the versions of Sable is 256, which is the same as the secret payload (m'). Therefore, we use one coefficient for a single bit of secret payloads and $B = 1$. The function $\texttt{arrange\_msg} : \{0,\ 1\}^{256} \longrightarrow \{0,\ 1\}^{256}$ is $\texttt{arrange\_msg}(m') = m' = m$ and the function $\texttt{original\_msg} : \{0,\ 1\}^{256} \longrightarrow \{0,\ 1\}^{256}$ is $\texttt{arrange\_msg}(m'') = m'' = m'$.

Secret Distribution

Like Florete, a coefficient of the secret vector is sampled from the CBD β_2 in every version of Sable. Therefore, one secret coefficient can be stored as a 2 bits number, allowing Sable to have smaller secret keys. All these choices help Sable to reduce stack memory requirements when implemented in a microcontroller, and area requirements when implemented in hardware. More details regarding implementation are provided in Sec. 5 & 6.

4 PARAMETER SET

The lattice-based schemes whose hardness depends on the LWE problems or its variant, such as the (M/R)LWR problem, are solved by utilizing lattice reduction algorithms that construct a "sufficient orthogonal" basis from the given lattice. Currently the best-known algorithm for lattice reduction is the BKZ algorithm. Here, given one lattice or a basis of lattice, the attacker needs to find the block size or sub-lattice size required for recovering the shortest vector of the lattice while performing the BKZ algorithm. The security of a lattice-based scheme depends on the cost of the execution time of the BKZ algorithm on the underlying lattice. The BKZ algorithm also calls the shortest vector problem (SVP) solving oracle on sub-lattices. The cost of solving the LWE problem with block size β depends on the number of SVP oracle calls made by the BKZ algorithm and the cost of solving each SVP for dimension β. This cost is approximately $2^{c\beta+o(\beta)}$ [6], where the value of c is approximately 0.292 in classical settings and 0.265 with Grover's speed-up algorithm [52] in quantum settings.

Dachman-Soled et al. [38] has introduced leaky-LWE-Estimator, the state-of-the-art toolkit to estimate the hardness of the underlying LWE problem for lattice-based schemes. This tool takes the n = dimension of the lattice, q = modulus, D_e = error distribution, D_s = secret distribution, and outputs the block size β. We have utilized this toolkit for the security estimation of our schemes. The post-quantum bit security is estimated as $0.265 * \beta$ [5], and classical bit security is estimated as $0.292 * \beta$ [15]. Since the post-quantum security is lower than classical security ($0.292 * \beta$ bit secure, we have mentioned only the post-quantum (PQ) security of our schemes in Table 3.

We present the parameter sets of our schemes for three security levels in Table 3. For security level 1, PQ security of each of the KEMs is $\geq 2^{100}$, for security level 3, PQ security is $\geq 2^{128}$, and for security level 5, PQ security is $\geq 2^{160}$. LWE-based cryptosystem has another security factor which is failure probability. However, another type of attack is possible on LWR or LWR-based cryptographic schemes that exploit failure probability during decryption. As mentioned in Sec. 2.3, the failure probability should be $\leq 2^{-S}$, where S is the security of the KEM to maintain the IND-CCA security of the KEM. Therefore, for security level 1, 3, 5 in contrast with the PQ security, the failure probability we maintain $\leq 2^{-100}$, $\leq 2^{-128}$, $\leq 2^{-160}$, respectively for each of the KEMs. NIST security levels 1, 3, and 5 are represented by low, medium, and high-security levels in Table 3.

Table 3. Compare parameters and key sizes of Scabbard suite with Saber

Scheme Name	Security level	Ring/Module Parameters		PQ Security	Failure probability	Moduli		CBD (β_η)	Encoding	Key sizes for KEM (Bytes)	
Florete	Low	n:	512	2^{104}	2^{-138}	ϵ_q:	11	$\eta = 1$	B=1	Public key:	608
						ϵ_p:	9			Secret key:	800
		l:	1			ϵ_t:	2			Ciphertext:	768
	Medium	n:	768	2^{157}	2^{-131}	ϵ_q:	10	$\eta = 1$	B=1	Public key:	896
						ϵ_p:	9			Secret key:	1152
		l:	1			ϵ_t:	3			Ciphertext:	1248
	High	n:	1024	2^{220}	2^{-165}	ϵ_q:	10	$\eta = 1$	B=1	Public key:	1184
						ϵ_p:	9			Secret key:	1504
		l:	1			ϵ_t:	4			Ciphertext:	1792
Espada	Low	n:	64	2^{101}	2^{-148}	ϵ_q:	15	$\eta = 3$	B=4	Public key:	1072
						ϵ_p:	13			Secret key:	1456
		l:	10			ϵ_t:	2			Ciphertext:	1088
	Medium	n:	64	2^{128}	2^{-167}	ϵ_q:	15	$\eta = 3$	B=4	Public key:	1280
						ϵ_p:	13			Secret key:	1728
		l:	12			ϵ_t:	3			Ciphertext:	1304
	High	n:	64	2^{168}	2^{-162}	ϵ_q:	15	$\eta = 3$	B=4	Public key:	1592
						ϵ_p:	13			Secret key:	2136
		l:	15			ϵ_t:	5			Ciphertext:	1632
Sable	Low	n:	256	2^{104}	2^{-139}	ϵ_q:	11	$\eta = 1$	B=1	Public key:	608
						ϵ_p:	9			Secret key:	800
		l:	2			ϵ_t:	2			Ciphertext:	672
	Medium	n:	256	2^{169}	2^{-143}	ϵ_q:	11	$\eta = 1$	B=1	Public key:	896
						ϵ_p:	9			Secret key:	1152
		l:	3			ϵ_t:	4			Ciphertext:	1024
	High	n:	256	2^{203}	2^{-208}	ϵ_q:	11	$\eta = 1$	B=1	Public key:	1312
						ϵ_p:	10			Secret key:	1632
		l:	4			ϵ_t:	2			Ciphertext:	1376
Saber	Low	n:	256	2^{107}	2^{-120}	ϵ_q:	13	$\eta = 5$	B=1	Public key:	672
						ϵ_p:	10			Secret key:	992
		l:	2			ϵ_t:	2			Ciphertext:	736
	Medium	n:	256	2^{172}	2^{-136}	ϵ_q:	13	$\eta = 4$	B=1	Public key:	992
						ϵ_p:	10			Secret key:	1440
		l:	3			ϵ_t:	3			Ciphertext:	1088
	High	n:	256	2^{236}	2^{-165}	ϵ_q:	13	$\eta = 3$	B=1	Public key:	1312
						ϵ_p:	10			Secret key:	1760
		l:	4			ϵ_t:	5			Ciphertext:	1472
Kyber	Low	n:	256	2^{107}	2^{-139}	q:	3329	$\eta_1 = 3$	B=1	Public key:	800
						ϵ_p:	10			Secret key:	1632
		l:	2			ϵ_t:	3	$\eta_2 = 2$		Ciphertext:	768
	Medium	n:	256	2^{166}	2^{-164}	q:	3329	$\eta_1 = 2$	B=1	Public key:	1184
						ϵ_p:	10			Secret key:	2400
		l:	3			ϵ_t:	3	$\eta_2 = 2$		Ciphertext:	1088
	High	n:	256	2^{232}	2^{-174}	q:	3329	$\eta_1 = 2$	B=1	Public key:	1568
						ϵ_p:	10			Secret key:	3168
		l:	4			ϵ_t:	4	$\eta_2 = 2$		Ciphertext:	1568

For comparing the key sizes of our schemes with Saber and Kyber, we also include the parameter sets of Saber in Table 3. The public key and secret key sizes of Florete are smaller than Saber, while the size of the ciphertext is slightly larger for Florete than Saber for all three security levels. The public key and secret key sizes of Florete are also smaller than Kyber for all three security versions. Even, the size of the ciphertext is the same for the low-security version of Florete and Kyber. Due to larger moduli and vector dimensions, the public key, secret key, and ciphertext sizes are bigger in Espada than in Saber for the same security level. However, the secret key size of Espada is smaller in Espada than in Kyber for all the security versions. In the case of any of the three security levels of Sable, the sizes of the public key, secret key, and ciphertext are smaller than in the case of the same security level of Saber and Kyber.

5 SOFTWARE IMPLEMENTATION

In this section, we describe the implementation results of our schemes on the software platforms. We have implemented Scabbard's schemes on general-purpose intel processors using C and advanced vector instructions (AVX2). We also implemented Scabbard's schemes on the NIST-recommended ARM Cortex-M4 platform. As most of the PQC schemes are implemented in these two software platforms, we can compare the implementation results of our schemes with the state-of-the-art schemes and demonstrate the efficiency of our scheme.

5.1 Results in C and AVX2

To implement our schemes on general-purpose intel processors using C and advanced vector instructions (AVX2), we use the GCC 6.5 compiler and optimization flags-O3. We also used -fomit-frame-pointer on an Intel (R) Core (TM) i7-6600 CPU running in 2.60GHz, and disabled hyperthreading, turbo-boost, and multicore support in our system following the standard practice. The performance results of Scabbard's schemes in portable C and AVX2 implementations are presented in Table 4. For comparison, we also include the performances of C and AVX2 implementation of NIST's third-round finalist Saber and NIST's standard Kyber together with BSI recommended [32] Frodo [29] (also in consideration of ISO for standard [53]) in the tables. This table also compares KPQC schemes Smaug [35], NTRU+ [69], and Tiger [90], which have been advanced to the second-round [73].

As we can see from Table 4, the performance of all three algorithms (key generation, encapsulation, and decapsulation) of Florete and Sable is better than all the other schemes, including Kyber, Frodo, and Saber, for all the security versions on C. All three algorithms of Florete and Sable perform better than Saber and Smaug in the AVX2 implementation as well for all the security versions. Algorithms of Espada take approximately twice as many clock cycles as Saber for all the security levels due to the use of more pseudo-random numbers and more 64×64 multiplications. However, the slowdown factor for Espada's performance compared to Saber's decreases as the security order increases because the underlying lattice rank $l \times n$ decreases in Espada compared to Saber as the security increases.

5.2 Results in Cortex-M4

We have implemented Scabbard's schemes on the NIST-recommended 32-bit ARM Cortex-M4 microcontroller (STM32F407-DISCOVERY development board) using the PQM4 [67] framework. For compilation, we have used `arm-none-eabi-gcc` compiler version 4.9.3. The PQM4 library uses a 24 MHz system clock to calculate clock cycles. The results of the implementations of Scabbard's schemes in a Cortex-M4 platform are presented in Table 5. We have also included the clock cycles spent in hashing, polynomial multiplication, and the remaining operations in this table. We have compared our implementations of Scabbard with the state-of-the-art schemes in Table 6. This table contains two implementations of Saber, one with NTT multiplication (SaberNTT [14]) and another with Toom-cook multiplication (Saber [14]). For

Table 4. Comparing performance of Scabbard schemes with Saber and kyber in portable C and AVX2 implementations

Scheme Name	Security level	C (X1000 clock cycles)			AVX (X1000 clock cycles)		
		KeyGen	Encaps	Decaps	KeyGen	Encaps	Decaps
Florete	Low	43	66	83	31	43	47
	Medium	64	104	143	45	66	75
	High	80	140	181	52	83	97
Espada	Low	159	173	193	148	158	153
	Medium	224	234	232	203	215	210
	High	336	351	352	310	324	317
Sable	Low	55	69	76	38	45	41
	Medium	101	126	137	63	74	71
	High	173	230	226	97	113	110
Saber [14]	Low	64	81	92	44	51	49
	Medium	116	143	154	73	85	82
	High	188	222	244	107	122	120
Kyber [27]	Low	113	150	176	26	38	29
	Medium	185	238	268	41	53	40
	High	301	341	382	49	66	51
Frodo [29]	Low	1237	1382	1383	-	-	-
	Medium	2654	2819	2509	-	-	-
	High	4225	4238	4465	-	-	-
Smaug [35]	Low	89	83	93	48	37	47
	Medium	157	148	158	73	59	74
	High	251	251	267	127	115	128
NTRU+ [69]	Low	339	110	164	18	15	12
	Medium	335	154	233	16	18	16
	High	358	180	277	14	20	18
Tiger [90]	Low	89	80	78	-	-	-
	Medium	104	122	127	-	-	-
	High	123	162	176	-	-	-

a fair comparison, we have also included implementation results of two versions of Kyber [67] and Frodo [28] (i) Kyber-Speed & Frodo-Speed: optimized to achieve speed, and (ii) Kyber-Stack & Frodo-Stack: optimized to reduce stack memory usage.

We have used optimized Toom-Cook-based polynomial multiplication for Florete, Espada, and Sable. These optimized polynomial multiplications are generated using the software package provided by Kannwischer et al. [66]. It can generate optimized assembly code for different combinations of Toom-Cook-based polynomial multiplications. As mentioned earlier, LWR-based schemes directly cannot use NTT. Later, Chung et al. [36] showed that Saber could use the NTT-based polynomial multiplication over a big prime field so that the absolute magnitude of the largest possible number occurs from the polynomial multiplication is smaller than the big prime. [36] also showed that Saber with NTT-based polynomial multiplication (SaberNTT) performs better than Saber with Toom-Cook-based polynomial multiplication. Afterward, Abdulrahman et al. [1] further improved the NTT-based polynomial multiplication of Saber using multi-moduli NTT. To show that Scabbard's schemes can be optimized for speed using NTT-based polynomial multiplication, we have implemented a version of Sable that uses NTT-based polynomial multiplication. We have

Table 5. Spent cycles of Scabbard schemes in hashing, polynomial multiplication, and other operations on the Cortex-M4 platform.

Scheme Name	Security level	Total Performance on M4 (x1000 clock cycles)			Hashing (x1000 clock cycles)			Polynomial multiplication (x1000 clock cycles)			Other components (x1000 clock cycles)		
		KeyGen	Encaps	Decaps	KeyGen	Encaps	Decaps	KeyGen	Encaps	Decaps	KeyGen	Encaps	Decaps
Florete	Low	299	536	606	158	261	183	121	243	364	20	32	59
	Medium	439	815	957	209	362	259	202	405	607	28	48	91
	High	598	1,131	1,357	259	463	335	300	600	900	39	68	122
Espada	Low	1,659	1,859	1804	1,029	1,170	1,054	482	530	579	148	159	171
	Medium	2,342	2,566	2,497	1,442	1,596	1,455	694	752	810	206	218	232
	High	3,577	3,859	3,779	2,181	2,372	2,206	1,085	1,157	1,230	311	330	343
Sable	Low	381	558	568	205	296	218	148	222	296	28	40	54
	Medium	745	1,005	1,031	363	491	388	333	444	555	49	70	88
	High	1,251	1,593	1,622	583	749	608	592	741	889	76	103	125
SableNTT	Low	306	431	419	204	294	218	66	94	132	36	43	69
	Medium	568	742	730	370	497	395	129	167	222	69	78	113
	High	924	1,149	1,124	599	763	624	213	260	331	112	126	169

implemented a multi-moduli NTT-based Sable with the help of the multi-moduli NTT-based implementation of Saber. We call this NTT-based Sable as SableNTT in Table 6.

We can observe from Table 6 that the KeyGen algorithm of Florete performs 34%, 49%, 57% faster than Saber, 31%, 38%, 47% faster than Kyber-Speed, and 99.6%, 99.7%, 99.8% faster than Frodo-Speed for low, medium, and high-security versions, respectively. The Encaps algorithm of Florete performs 15%, 26%, 33% faster than Saber, and 99.4%, 99.6%, 99.7% faster than Frodo-Speed for low, medium, and high-security versions, respectively. The Encaps algorithm of Florete performs 6%, 14% better compared to Kyber-Speed for medium, and high-security versions, respectively. The Decaps algorithm of Florete performs 6%, 15%, 21% faster than Saber, and 99.3%, 99.5%, 99.6% faster than Frodo-speed for low, medium, and high-security versions, respectively. However, one thing to note is that the improvement of the performance of the KeyGen, Encaps, and Decaps algorithms for Florete against Saber increases as security increases. Also, the performance improvement of the KeyGen, Encaps algorithms for Florete against Kyber-Speed increases as security increases, and the slowdown factor for the Decaps algorithm of Florete against Kyber decreases as security increases.

The stack memory requirements for the implementations of low, medium, and high-security versions of the KeyGen algorithm in Espada are respectively 58%, 56%, 52% lower than Saber and 68%, 47%, 50% lower than Frodo-Stack. The KeyGen algorithm of Espada requires more stack memory than Kyber[1]. For implementations of low, medium, and high-security versions of the Encaps algorithm in Espada, the stack memory requirements are respectively 67%, 68%, 67% lower than Saber, 16%, 26%, 30% lower than Kyber, and 71%, 56%, 57% lower than Frodo-Stack. The stack memory requirements in implementations of low, medium, and high-security versions of the Decaps algorithm in Espada are respectively 69%, 69%, 68% lower than Saber, 22%, 30%, 34% lower than Kyber, and 72%, 56%, 58% lower than Frodo-Stack.

The KeyGen algorithm of Sable performs at least 9% faster than Saber and at least 99.5% faster than Frodo-Speed. The Encaps and Decaps algorithms of Sable perform at least 6% faster than Saber and at least 99.3% faster than Frodo-Speed. All the algorithms of Sable also use less stack memory compared to Saber for all the security versions. The NTT-based

[1] The latest Kyber uses a different technique to generate matrix A and A^T during the matrix-vector multiplication than Espada. In Kyber, each polynomial of the matrix A and A^T can be generated independently from SHAKE-128 with a slightly different version of the seed. Therefore, each polynomial of the matrix A can be generated run-time during matrix-vector multiplication using just-in-time strategy [68] and only one polynomial space is required for the matrix A and A^T. However, in Espada, the whole matrix A is generated from a single seed. So, between matrix A and A^T, the matrix A can utilize the maximum benefit from the just-in-time strategy. Here, the matrix A needs one polynomial to store the whole matrix, but the matrix A^T needs a vector of polynomial space for the entire matrix. Therefore, the KeyGen algorithm of Espada requires more stack memory than Kyber. However, Kyber's matrix generation technique can also be used in Espada. Then, like Encaps algorithm, KeyGen algorithm of Espada will require less stack memory than Kyber.

Table 6. Comparing performance and stack memory requirement of Scabbard schemes with Saber and Kyber on Cortex-M4 platform

Scheme Name	Security level	Performance (x1000 clock cycles)			Stack memory (bytes)		
		KeyGen	Encaps	Decaps	KeyGen	Encaps	Decaps
Florete	Low	299	536	606	8,256	8,392	8,392
	Medium	439	815	957	18,252	18,420	18,420
	High	598	1,131	1,357	25,408	25,608	25,608
Espada	Low	1,659	1,859	1804	2,544	1,960	1,840
	Medium	2,342	2,566	2,497	2,896	2,120	2,000
	High	3,577	3,859	3,779	3,424	2,360	2,240
Sable	Low	381	558	568	5,672	5,928	5,432
	Medium	745	1,005	1,031	6,184	5,992	5,496
	High	1,251	1,593	1,622	6,696	6,056	5,560
SableNTT	Low	306	431	419	5,548	6,220	6,228
	Medium	568	742	730	6,564	7,244	7,252
	High	924	1,149	1,124	7,596	8,276	8,284
Saber [14]	Low	454	631	643	6,060	6,020	6,028
	Medium	856	1,106	1,121	6,572	6,540	6,548
	High	1,382	1,694	1,726	7,084	7,052	7,060
SaberNTT [1]	Low	351	481	452	5,628	6,308	6,316
	Medium	644	820	773	6,652	7,332	7,340
	High	992	1,203	1,149	7,676	8,348	8,356
Kyber-Speed [67]	Low	434	530	477	4,320	5,424	5,432
	Medium	707	863	783	5,344	6,456	6,472
	High	1,123	1,316	1,210	6,400	7,496	7,512
Kyber-Stack [67]	Low	434	532	478	2,248	2,336	2,352
	Medium	707	867	788	2,784	2,856	2,872
	High	1,127	1,324	1,219	3,296	3,368	3,392
Frodo-speed [28]	Low	75,000	85,000	84,000	12,516	14,468	14,476
	Medium	169,000	186,000	185,000	18,572	19,860	19,868
	High	309,000	345,000	344,000	25,196	25,764	25,772
Frodo-stack [28]	Low	223,000	293,000	294,000	7,948	6,668	6,460
	Medium	1,103,000	1,296,000	1,296,000	5,444	4,796	4,596
	High	2,003,000	2,380,000	2,379,000	6,916	5,532	5,324

version of Sable is named SableNTT. The `KeyGen` algorithm of SableNTT performs 13%, 12%, 7% faster than SaberNTT, 29%, 20%, 18% faster than Kyber-Speed, and 99.6%, 99.7%, 99.7% faster than Frodo-Speed for low, medium, and high-security versions, respectively. The `Encaps` algorithm of SableNTT performs 10%, 10%, 4% faster than SaberNTT, 19%, 14%, 13% faster than Kyber-Speed, and 99.5%, 99.6%, 99.7% faster than Frodo-Speed for low, medium, and high-security versions, respectively. The `Decaps` algorithm of SableNTT performs 7%, 6%, 2% faster than SaberNTT, 12%, 7%, 7% faster than Kyber-Speed, and 99.5%, 99.6%, 99.7% faster than Frodo-speed for low, medium, and high-security versions, respectively. Also, all three algorithms of SableNTT also need less stack memory compared to SaberNTT for all the security versions.

6 HARDWARE IMPLEMENTATION

The following section describes our design decisions for the unified implementations of Scabbard (medium security version) in hardware, followed by a discussion and comparison of the results. We will demonstrate how taking hardware efficiency into account during the design cycle of cryptographic schemes leads to efficient implementations on hardware platforms.

All schemes in the Scabbard suite (Florete, Espada and Sable) are LWR-based KEMs and consist of three main routines: (i) key-generation (`LWR.KEM.KeyGen`), (ii) encapsulation (`LWR.KEM.Encaps`), and (iii) decapsulation (`LWR.KEM.Decaps`). As outlined in Sec. 2 and 3, these share several fundamental operations, including polynomial multiplication, hashing, pseudo-random number generation, and binomial sampling. We explore the trade-off between high speed, low area and high flexibility for the full-hardware implementation of the KEM operations by (re-)using common building blocks where possible. Where possible, we optimize our implementation to meet our design objectives, outlined in Sec. 3.

6.1 High-level Architecture

We follow a hardware (HW) only design methodology as opposed to a HW/SW co-design strategy. In a HW/SW co-design, only the most computationally expensive operations (i.e. polynomial multiplier) are implemented on hardware, providing high flexibility at the cost of reduced performance. In our implementations, all the building blocks reside in hardware, as we prioritize speed. Yet, our implementation remains flexible (e.g. support for key generation, encapsulation, and decapsulation) by targeting an *instruction-set coprocessor architecture* (ISA), as proposed in [100]. Such a unified architecture offers instruction-level flexibility and modularity. A high-level diagram of the ISA for all schemes of Scabbard is shown in Fig. 5. As the CCA-secure KEM routines for Florete, Espada, and Sable follow a common framework (Sec. 2.3), the high-level architecture of their hardware implementations are similar. The differences between implementations of sub-blocks of different schemes are explicitly listed, and our design methodology is explained, if applicable. We also focus on the particularities of the different polynomial multipliers for Florete, Espada, and Sable.

The coprocessor is controlled by loading the program memory with the microcode of the protocol (e.g. key generation). The instruction words are 35-bit, consisting of a 5-bit wide instruction code, and 3×10-bit data addresses, of which two are for input operands and one for the result. The algorithms and instructions are designed to not include conditional branching, to prevent timing-based side-channel attacks. The main communication controller interacts with the individual building blocks, which are designed to be constant-time. As a result, each of the implementations of all KEM operations takes a fixed amount of time.

6.2 Data Memory

Both input data and results for each of the operations are read from/written back to the data memory, which is implemented using BRAM tiles. The medium security versions of all Scabbard schemes require at minimum an 8KB memory size such that all KEM routines can be computed. The word size is 64-bit, as this allows for easy integration of the ISA co-processor with a 32-bit or 64-bit host computer. We ensure data is optimally packed inside the 64-bit words, and all individual blocks maximally exploit this format.

6.2.1 Espada. In order to be able to store the public matrix $\mathbf{A}$, generated from the public seed$_A$ using an XOF, we instantiate an additional ± 17KB of *parallel* data memory. We design and implement it to consist of $l = 12$ parallel memory banks, which each store $n = 64$ coefficients. It allows for our polynomial multiplier to maximally exploit its

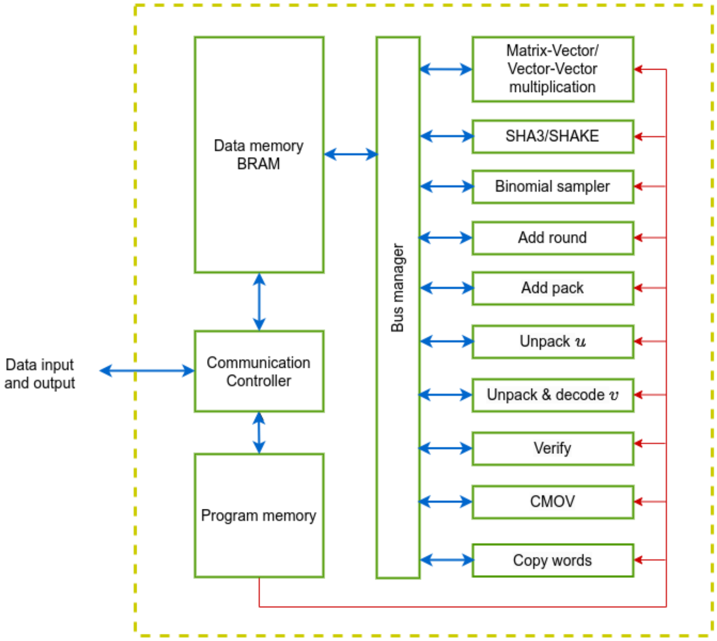

Fig. 5. The high-level architecture diagram of the instruction set processor for schemes of Scabbard. The blue line symbolizes the data bus, and the red line indicates the control signal.

parallel nature by simultaneously reading from and writing to all l memory banks in a Single-Instruction Multiple-Data (SIMD) fashion during matrix-vector multiplication (Fig. 7).

6.3 SHA3/SHAKE

The Scabbard suite relies on the HW-friendly Keccak sponge function (FIPS 202) [44] through the hash functions SHA3-256 and SHA3-512 and the extendable output function SHAKE-128 for generating pseudorandom numbers. The SHA3/SHAKE block is implemented using the open-source high-speed implementation of the Kecak core, designed by the Keccak Team [102]. Around this sits the SHA3/SHAKE wrapper from the open-source implementation of Saber on hardware [100].

All data padding and extraction operations are performed in the wrapper in hardware, controlled by a second instruction from the program memory. The input/output data length is flexible and first specified through 2×16-bit fields, followed by the data and result operand addresses. The SHA3/SHAKE block consumes around 5,900 LUTs and 3,127 FFs, or up to 35% of total area utilization of the full HW implementation of Sable. Also, during the decapsulation operation, up to 21% of total execution time (1,521 clock cycles for Sable) is required for Keccak-related operations. For Florete, where the polynomial multiplier is a more performance-critical component, the SHA3/SHAKE block accounts for around 20% of total area utilization. For a full decapsulation operation, 4% of total execution time is required for Keccak-related operations. We argue that instantiating a single Keccak core in hardware is a good compromise, as we achieve high speed, and this building block is already an area-expensive component.

6.3.1 Florete. As this scheme is based on the RLWR hard problem, the ring/modulus parameter l is equal to 1. The public matrix $\mathbf{A}$ is a polynomial and an element of the ring $\mathcal{R}_q^n$. Compared to Sable and Espada, where $\mathbf{A}$ is a matrix, the generation of this public value is much cheaper for Florete in hardware. Only 426 clock cycles are required in total for all SHAKE-128 operations during encapsulation and decapsulation.

6.4 Binomial Sampler

In Scabbard, the secret coefficients are drawn from a centered binomial distribution with parameter μ. A μ-bit pseudo-random string $r[\mu - 1 : 0]$ is split in two parts, and the Hamming weight `HW()` of each is subtracted. More specifically, $\texttt{HW}(r[\frac{\mu}{2} - 1 : 0])\ \texttt{-}\ \texttt{HW}(r[\mu - 1 : \frac{\mu}{2}])$ is computed. As proposed in [100], the sampler is implemented as a combinatorial block with an input and output buffer. The output samples are in sign-magnitude representation.

6.4.1 Florete & Sable. For both schemes $\mu = 2$, meaning the secret coefficients are in $[-1 : 1]$ (two bits), and 32 samples can be directly stored in a 64-bit data word. First, a 64-bit pseudo-random word is loaded from the data memory, stored in a buffer and then 32 samples are generated in parallel. The 64-bit result is transferred from the output buffer to the global data memory and repeated until the full secret polynomial is generated.

6.4.2 Espada. As $\mu = 6$, which is not a divisor of 64, the input buffer is 192-bit since $\texttt{lcm}(6, 64) = 192$. Three 64-bit, pseudo-random strings are loaded to the input registers, after which 16 4-bit samples are computed twice in a row. Generating 16 output samples requires $96 = 2 \cdot 3 \cdot 16$ bits, meaning the process is repeated twice until the input buffer is filled again.

6.5 Polynomial Multiplication

The following section discusses the particularities of our polynomial multiplier design for Florete, Espada, and Sable. As Scabbard was designed to be polynomial arithmetic-friendly, we use off-the-shelf and state-of-the-art polynomial multipliers to demonstrate their efficiency in hardware. We integrate the multipliers in our high-speed hardware design and modify them so they support both inner-product and matrix-vector polynomial multiplications on the same hardware. Depending on the instruction loaded from the program memory, the appropriate control signals are set, and the operation is performed.

During operation, typically, the secret polynomial $\mathbf{s}$ is first loaded from the data memory, unpacked, and stored in a LUT-based buffer. Secondly, the polynomial multiplicand $\mathbf{a}$ is loaded into an input buffer. For the matrix-vector multiplication $\mathbf{A} \cdot \mathbf{s}$ or $\mathbf{A}^T {\cdot} \mathbf{s}$, the coefficients are ϵ_q bits and generated by `SHAKE-128`, 64 bits at a time and continuously stored in the data memory. The input buffer unpacks the coefficients which may be split across different words in data memory, to 16-bit operands for the multipliers. The processing starts as soon as the first few coefficients are available in this buffer, parallelizing the computation and data transfers, as proposed in [100]. For the inner-product calculations $\mathbf{u}^T {\cdot} \mathbf{s}$ or $\mathbf{b}^T {\cdot} \mathbf{s}'$, the coefficients of the polynomial multiplicand are ϵ_p-bit wide and zero-padded up to (and stored as) 16-bit coefficients. Four are loaded from the data memory at once (in one 64-bit word) and directly stored in the polynomial multiplicand registers. After the computation has finished, the coefficients of the resulting polynomial are zero-padded to 16-bit, packed into 64-bit data words, and stored in the data memory.

6.5.1 Florete. In Florete, a 768×768 polynomial multiplication is required, which we decompose into smaller (256×256) polynomial multiplications by implementing Toom-Cook 3-way evaluation and interpolation in hardware. These acts as a wrapper around five 256×256 polynomial multipliers, which we all instantiate in parallel. A benefit of our approach, using Toom-Cook 3, is that our implementation can reuse any state-of-the-art 256×256 polynomial multiplier available in literature [39, 83, 100].

In order to further exploit parallelism in our hardware implementation, we break down the polynomial multiplication further using Toom-Cook 4-way evaluation and interpolation as proposed in [83]. Hence, each 256×256 polynomial multiplier consists of 7 parallel 64×64 polynomial multipliers, which all execute in parallel. In total, $7 * 5 = 35$ 64×64 polynomial multiplications are instantiated and performed in parallel (Fig. 6).

As a result, a 768×768 multiplication takes as long as a single 256×256 multiplication at the cost of a five times larger area. In between the different stages, intermediate and accumulated results are stored in LUT-based buffers. The evaluation and interpolation datapath are pipelined.

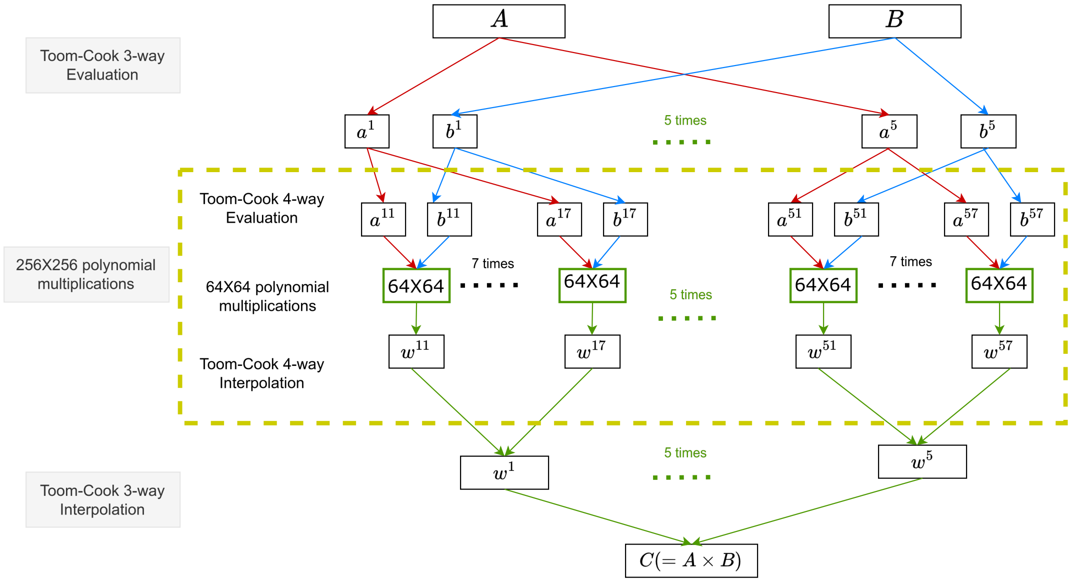
A
B
Toom-Cook 3-way Evaluation
a1
b1
5 times
a5
b5
Toom-Cook 4-way Evaluation
a11
b11
a17
b17
a51
b51
a57
b57
256X256 polynomial multiplications
64X64 polynomial multiplications
7 times
64X64
64X64
5 times
7 times
64X64
64X64
w11
w17
w51
w57
Toom-Cook 4-way Interpolation
w1
5 times
w5
Toom-Cook 3-way Interpolation
C(= A × B)

Fig. 6. Polynomial multiplication of Florete. 768×768 multiplication is decomposed into 35 64×64 polynomial multiplications, using Toom-Cook 3-way and Toom-Cook 4-way.

6.5.2 Espada. Our Espada multiplier is designed to exploit the inherent parallelism of the scheme and its matrix-vector multiplication. We do not require any evaluation/interpolation steps to break down a large polynomial multiplication due to the choice of parameters and choice of module lattices. More specifically, as can be observed in Fig. 7, l 64×64 polynomial multipliers are instantiated in parallel ($l = 12$ for medium security level). During the matrix-vector multiplication, each multiplier is fed with one row of the public matrix of dimension $l \times l$ in parallel. During the inner-product operation, only one multiplier is active. In both cases, the corresponding secret polynomial is the same for all multipliers and is loaded first in a small LUT-based buffer.

Before the computation starts, the polynomial multiplicands are loaded to small LUT-based buffers, instantiated for each of the multipliers. These allow for each of the multipliers to perform read and write operations during computation. In order to minimize the overhead of loading the large public matrix $\mathbf{A}$ from data memory and writing back the results of all l multipliers to global memory, we instantiate an additional data memory consisting of $l = 12$ banks. During the generation of the public matrix using SHAKE-128, these are filled in a sequential manner, containing one row of the matrix each. The multipliers read the polynomial multiplicands from these banks in parallel and write their results to l banks in parallel. As such, not only the computation but also the read and write operations are parallelized (on the matrix-vector level), bringing the performance close to state-of-the-art.

For both Florete and Espada, our architecture leaves space to optimize the 64×64 multipliers for area or performance. Our parametrizable design allows us to select the number of arithmetic units (implemented using DSP units), achieving higher performance at the cost of higher area utilization. We choose 4 DSP units per polynomial multiplier, in order to keep area cost at a reasonable level. As a result, our latency is high compared to our Sable implementation, as our multiplier requires $16 * 64$ clock cycles to perform a full 64×64 multiplication.

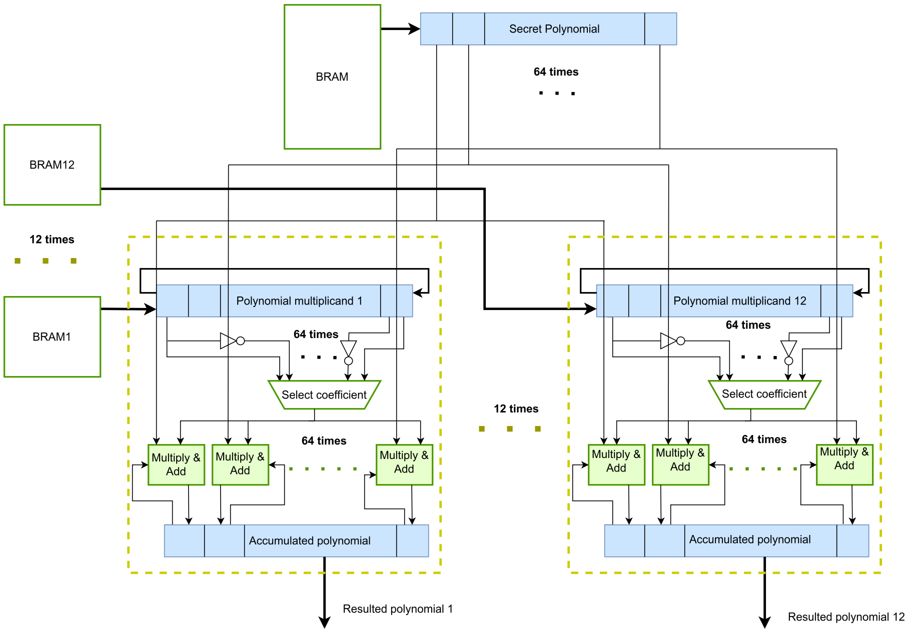

Fig. 7. Polynomial multiplication for Espada. Blue colored blocks represent register and the multiplication and add block is colored green. The dotted yellow colored block represents a 64 × 64 polynomial multiplication. Here, we perform 12 such multiplication in parallel.

6.5.3 Sable. The Sable multiplier is based on the high-speed Saber implementation in [100] but optimized for the Sable parameters. As a result, the area requirements are reduced without a performance loss as the property of 2-bit secrets is exploited. Our proposed architecture is drawn in Fig. 8. The custom 'Multiply & Add' arithmetic unit (in green) is instantiated 256 times, resulting in a full parallel polynomial multiplication. The arithmetic unit is a combinatorial block, thus, a full $N = 256$ polynomial multiplication requires only 256 cycles.

Before computation, the entire 512-bit secret polynomial $\mathbf{s}$ is loaded in a LUT-based shift register, which allows for the negacyclic convolution to be performed in-place and access to all secret coefficients at once. The nega-cyclic left-shift operation moves each secret coefficient from position i to $i + 1$ and the last secret to the first position after modular subtraction from zero. As the secret coefficients use sign-magnitude representation, only a simple sign flip is required.

The full polynomial multiplicand **a** is also loaded into an input buffer, from which four coefficients at a time are processed by the arithmetic units. For matrix-vector multiplication, as $\epsilon_q = 11$, 11-bit coefficients are unpacked to four 16-bit coefficients by the input buffer, and 64-bit data words are fed to the MAC units. For the inner-product calculations, four zero-padded coefficients are stored in one data-memory word ($\epsilon_p = 11$), which are directly wired to the MAC units.

The result of the arithmetic unit is stored in the output accumulator buffer. This value is reset or preserved, depending on whether an inner product or full row-column multiplication (matrix-vector multiplication) is computed. Upon completion, the resulting polynomial is stored in the global data memory.

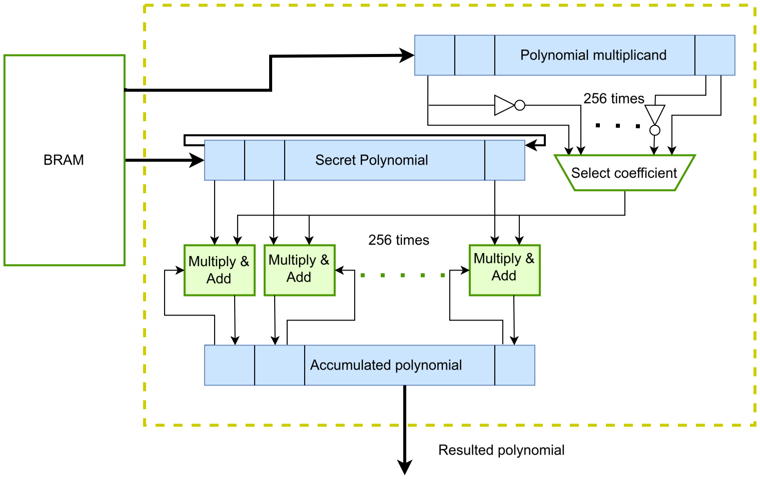

Fig. 8. Polynomial multiplication for Sable. Blue colored blocks represent register and the multiplication and add block is colored green.

The custom arithmetic unit is optimized for Sable's 2-bit secrets: only if $\texttt{LSB}(s_i)$ is 1 the accumulated result will be updated. The most significant bit of the secret determines if a_i is added or subtracted from the result.

6.6 AddRound, AddPack, and Unpack/Decode

All three schemes in the Scabbard suite use power-of-two moduli $p = 2^{\epsilon_p}$ and $q = 2^{\epsilon_q}$. In hardware, this translates to modular reduction and rounding being essentially free as they consist of shifting, re-wiring, adding, and bit-selecting. However, as the exact parameters are rarely selected as multiples of 8, low-level bit manipulations and small memory buffers are required. We fine-tune our implementation to minimize additional area utilization.

6.7 Remaining sub-blocks

The *Verify* module compares the received ciphertext and re-encrypted ciphertext during the decapsulation, word-by-word, and stores the result in a flag register. The *CMOV* module copies either the shared session key K or a pseudo-random string to a specified location based on this flag. The data move is constant-time.

6.7.1 Florete. As $l = 1$ for Florete, being based on the RLWR hard problem, $\mathbf{A}$ consists of a single polynomial. As a result, no matrix transpose is required during key generation, resulting reducing the cycle count.

6.7.2 Espada. An additional module, *CopyTranspose*, is added in order to efficiently transpose the $l \times l$ matrix $\mathbf{A}$. Still, it is relatively more expensive compared to other Scabbard schemes and Saber.

6.8 Performance Evaluation

Our full-hardware ISA is described in mixed Verilog and VHDL and compiled using Xilinx Vivado 2021.1 (default strategies) for the target platform Xilinx ZCU102 board, containing an Ultrascale+ XCZU9EG-2FFVB1156 FPGA and Arm Cortex-A53 host processor. Before a KEM operation, all operand data is transferred from the host processor to the coprocessor at once, then all computations are performed on the FPGA, and the result is read back by the host processor.

6.8.1 Timing Results. We first give a detailed breakdown of the cycle counts for the individual low-level operations and total cycle count in Table 7. Numbers for our implementation of all Scabbard schemes and the Saber implementation (using 256 MAC units & multipliers) from [100] are provided and compared. Table 8 shows the total execution times for our hardware implementations, calculated at 150MHz using Vivado simulation. We compare our designs of the Scabbard suite, which are based on variants of the LWR problem, with Saber, as it is the most well-known LWR-based scheme. For Keygen/Encaps/Decaps operations, our Sable implementation requires 13/11/10% fewer clock cycles. This is mainly due to our optimized multiplier design's relaxed requirements for sampling pseudo-random numbers compared to Saber. Our multiplier, similar to the Saber design, uses 256 MAC units and multipliers, which allows for the best direct comparison. It is clear that our design decisions lead to improved performance in hardware.

Additionally, all Scabbard schemes benefit from their choice of secret distribution, which results in more efficient vector sampling. For all KEM operations, Florete/Espada/Sable require 84/16/84% fewer clock cycles compared to Saber, respectively.

In all Scabbard KEM operations, the time spent performing polynomial multiplications is significant: 85/85/86%, 69/82/87% and 55/59/60% of total Keygen/Encaps/Decaps cycle counts. Our Sable multiplier is optimized for the 2-bit secrets and consists of 256 MAC units and multipliers, resulting in a low total latency.

Both Espada and Florete rely on 64×64 polynomial multipliers, which are implemented using only 4 DSP units. Increasing the DSP units of each multiplier will bring their performance closer to the state-of-the-art at the cost of increased area utilization. Our implementation prioritizes area cost while still achieving reasonable latency overhead. Notice that for Espada, due to our parallelized design, the latency for a complete matrix-vector multiplication and inner product are identical, as l polynomial multipliers are instantiated in parallel.

The second significant factor in the execution time of Scabbard are all Keccak-based functions: SHA3-256, SHA3-512, and SHAKE-128. For key generation, encapsulation, and decapsulation, this is 10/8/4%, 24/16/10%, and 28/28/21% of total cycle counts, respectively.

Compared to Saber, our Florete implementation requires 67/70/70% fewer clock cycles during Keygen/Encaps/Decaps operations. Our Sable HW implementation requires 22/24/24% fewer clock cycles compared to Saber. A large contributing factor to the Espada cycle counts is its randomness requirement (SHAKE-128) for the generation of $\mathbf{A}$ (around 5K clock cycles).

Table 7. Total cycles spent in low-level operations for Scabbard schemes and Saber [100] using Vivado simulation (Medium security parameters).

Instruction	Scheme Name	Cycle Count		
		KeyGen	Encaps	Decaps
SHA3-256	Florete	200	589	333
	Espada	272	661	333
	Sable	200	541	387
	Saber	339	585	303
SHA3-512	Florete	0	65	68
	Espada	0	65	68
	Sable	0	65	68
	Saber	0	62	62
SHAKE-128	Florete	489	426	426
	Espada	5,352	5,286	5,286
	Sable	1,135	1,066	1,066
	Saber	1,461	1,403	1,403
Vector sampling	Florete	28	28	28
	Espada	147	147	147
	Sable	25	28	28
	Saber	176	176	176
Polynomial multiplications	Florete	6,051	12,102	18,153
	Espada	15,826	31,652	47,478
	Sable	2,598	3,464	4,330
	Saber	2,685	3,592	4,484
Remaining operations	Florete	318	954	2,087
	Espada	1,441	767	1,511
	Sable	783	738	1,385
	Saber	792	800	1,606
Total cycles	Florete	7,086	14,164	21,095
	Espada	23,038	38,578	54,823
	Sable	4,741	5,902	7,264
	Saber	5,453	6,618	8,034

6.8.2 Area Results. In Table 9 a detailed breakdown of the area utilization of the full instruction-set coprocessor architecture of all Scabbard schemes and Saber is provided. We include numbers of the internal building blocks.

Our full HW Sable implementation requires 27% less LUTs compared to the state-of-the-art Saber implementation and 15% more FFs. The increase in registers is related to the choice of $\epsilon_p = 9$ and $\epsilon_q = 11$, which result in non-multiples of 8-bit operands. As a result, larger intermediate buffers are required to temporarily store data operands during conversion to 16-bit operands. Our flexible Espada implementation utilizes 20% fewer LUTs and requires 14 BRAM tiles

Table 8. Execution times for Scabbard schemes and Saber [100] using Vivado simulation calculated at 150 MHz (Medium security parameters).

Instruction	Scheme Name	Time (μs) KeyGen	Encaps	Decaps
Total time at 150 MHz	Florete	47.24	94.43	140.63
	Espada	153.59	257.19	365.49
	Sable	31.61	39.35	48.43
	Saber	36.35	44.12	53.56

(to store l rows of $\mathbf{A}$) instead of 2 in the Saber implementation. On the matrix-vector level, our Espada implementation is fully parallelized, consisting of $l = 12$ multipliers and intermediate buffers.

For all implementations, the poly-vector multiplier is the largest contributor at 75/66%, 39/62%, and 58/49% of total LUT/FF counts (and all DSP units) for Florete, Espada, and Sable, respectively. Still, due to making hardware-aware design choices, our multipliers perform well compared to prior art. Our Sable multiplier utilizes 43% fewer LUTs and only 9% more FFs due to its small secret coefficients and custom shift register architecture. Our Espada design requires 58% fewer LUTs and around two times as many FFs. This is due to the fact that each of the l multipliers requires intermediate buffers for the unpacking of public matrix $\mathbf{A}$ to 16-bit coefficients in parallel. Our Florete multiplier design consists of a full Toom-Cook 3 and 4-way evaluation and interpolation (pipelined) datapath, with 35 64×64 multipliers. Because of the massive parallelization, our implementation utilizes 21% more LUTs and 2 times more FFs compared to Saber. By choosing only 4 DSP units per multiplier, we keep the area utilization at a reasonable level. However, our flexible design allows to use of any 64×64 polynomial multiplier, including increasing the DSP units per multiplier.

6.8.3 Comparisons with existing implementations. Our high-speed and highly flexible architectures for Florete, Espada, and Sable are compared with recent hardware implementations of other post-quantum KEM schemes in Table 10. It is important to note that different hardware implementations target different schemes, security levels, platforms, or design methodologies. As a result, a fair and direct comparison is not always possible. The timing results of our implementation are derived from the Vivado simulation, calculated at 250MHz.

The fairest comparison is of our high-speed Sable implementation with the Saber implementation by [100]. Both are high-speed designs and implement the polynomial multiplier with 256 MAC units and multipliers, costing 256 cycles in total. Due to the hardware-aware design decisions, Sable requires 13/11/10% fewer clock cycles and has a lower area utilization.

Our Espada and Florete implementations are targeting a trade-off between high-speed and low-area. More specifically, the multiplier architecture around the 64×64 polynomial multipliers is highly parallelized and pipelined, allowing for efficient data transfers. We implement the 64×64 multipliers with only 4 DSP units per multiplier in order to reduce area utilization, requiring 64*16 clock cycles for complete multiplication. Our flexible design allows for the DSP count to be increased and directly reduces the total latency up to a factor of 16 or to be replaced with any available off-the-shelf designs.

Frodo KEM is implemented in hardware by How et al. [61] and uses dedicated data paths for `KeyGen`, `Encaps` and `Decaps`. The security of Frodo is based on the standard LWE problem, meaning the computationally expensive matrix-vector multiplications need to be computed several times. As a result, the latency of KEM operations is significantly

Table 9. Area results for full HW implementation of Scabbard schemes and Saber [100], with clock frequency constraint set to 250MHz in Vivado (Medium security parameters).

Block	Scheme Name	LUTs	FFs	DSPs	BRAMs
SHA3/SHAKE	Florete	5,834	3,126	0	0
	Espada	5,964	3,127	0	0
	Sable	5,831	3,127	0	0
	Saber	5,113	3,068	0	0
Binomial Sampler	Florete	79	86	0	0
	Espada	253	282	0	0
	Sable	67	86	0	0
	Saber	92	88	0	0
Poly-vector multiplier	Florete	21,143	10,613	140	0
	Espada	7,286	11,662	48	0
	Sable	9,841	5,523	0	0
	Saber	17,429	5,083	0	0
Other blocks	Florete	1,225	2,204	0	0
	Espada	5,238	3,752	0	0
	Sable	1,353	2,544	0	0
	Saber	1,052	1,566	0	0
Full co-processor	Florete	28,281	16,029	140	2
	Espada	18,741	18,823	48	14
	Sable	17,092	11,280	0	2
	Saber	23,686	9,805	0	2

higher compared to ring or module lattice-based schemes, like Scabbard or Saber. The high-speed Kyber implementation targets an extremely high operating frequency (450MHz), which translates into a faster execution time.

Compared to the high-speed NTRU prime hardware implementation by Peng et al. [40], Sable outperforms both in performance and area utilization. Florete achieves faster (total) execution time at lower area utilization, mainly due to the expensive KeyGen of NTRU Prime. Espada is slower yet has significantly lower area utilization, which is our design goal nonetheless. Compared to the low area NTRU prime hardware implementation, all Scabbard implementations significantly outperform NTRU performance-wise. However, this NTRU implementation has a lower area utilization than any other work listed in Table 10.

More recently, several designs tailored for ASIC have been published. Ghosh et al. [49] implemented NIST Round 3 Saber [14] in TSMC 65nm technology, targeting a low power consumption. Their crypto accelerator runs at 160 MHz and occupies 0.158 mm^2 and is 2.07/0.76/4.92 times slower compared to our implementations. We also highlight a unified Dilithium/Kyber ASIC implementation by Aikata et al. [2], occupying 0.263 mm^2 (TSMC 28 nm) and which utilizes multiple clock domains. As they utilize an advanced technology node, their design can run at 2 GHz (compared to 250 MHz) and still our Sable implementation is only 9% slower in total execution time.

Table 10. Overview and comparison of Scabbard schemes and existing hardware implementations of CCA-secure KEM schemes. (Medium security level.)

Implementation	Platform	Time in μs (KeyGen/Encaps/Decaps)	Frequency (MHz)	Area (LUT/FF/DSP/BRAM) (or mm^2 for ASIC)
Florete	UltraScale+	28.3/56.7/84.4	250	28.2K/16.0K/140/2
Espada	UltraScale+	92.2/154.3/219.3	250	18.7K/18.8K/48/14
Sable	UltraScale+	18.9/23.6/29.0	250	17.0K/11.2K/0/2
Saber [100]	UltraScale+	21.8/26.5/32.1	250	23.6K/9.8K/0/2
Kyber [40]	UltraScale+	5.9/8.3/10.9	450	10.6K/10.5K/6/6.5
Frodo [61]	Artix-7	45K/45K/47K	167	≈7.7K/3.5K/1/24
NTRU Prime (High Speed) [91]	UltraScale+	224.7/17.3/38.6	285	40.1K/26.4K/36.5/31
NTRU Prime (Low Area) [91]	UltraScale+	2.2K/100.8/302.6	285	9.2K/4.4K/8.5/18
Saber [49]	ASIC (65 nm)	89/117/146	160	0.158
Kyber [2]	ASIC (28 nm)	6.18/11.09/47.89	2K	0.263

From our experimental performance evaluation we can conclude that our Scabbard coprocessors have fast computation time compared to other lattice-based KEM hardware implementations, with moderate area utilization. In the case of Sable, which utilizes the same multiplier architecture as the Saber implementation in [100], our implementation requires 13%/11%/10% fewer clock cycles for KEM operations and lower area utilization. By considering hardware design choices during the design of the schemes themselves, efficient implementations have been achieved. Our full-hardware instruction-set coprocessor architectures result in high-speed and highly flexible implementations. We leave support for multiple parameter sets and security versions in a single hardware implementation as future work.

7 PHYSICAL ATTACK ANALYSIS

Physical attacks have been demonstrated to be very potent against even for mathematically secure cryptographic algorithms [26, 70–72], and lattice-based cryptography is no exception [10, 54, 63, 85, 94]. Therefore, physical attack analysis is one of the essential measures that ought to be carried out before deploying cryptographic algorithms in the real-world. Physical attacks can be divided into two categories depending on their properties: (i) passive attacks, which include timing-based side-channel attacks [54, 70], power-based side-channel attacks [10, 12, 63, 71, 85], side-channel attacks based on electromagnetic radiation (EM) [72, 94], etc., and (ii) active attacks, which include fault-injection attacks [26, 74, 84, 94].

Timing-based SCA is mitigated with constant-time implementations, where the execution time of the cryptographic algorithm does not depend on the secret data. All of our software and hardware implementations of Scabbard are in constant-time. It has been accomplished by avoiding secret-data-dependent operations (such as division), secret-data-dependent conditional operations (if-else), or secret-data-dependent memory access. Recently, Bernstein et al. [19] have proposed that the modular divisions used in Kyber's implementation are vulnerable to timing SCA due to division by prime modulus. These timing SCA do not apply to our schemes as our divisions are merely shift operations.

The implementations of Scabbard are susceptible to power- or EM-based SCA like other lattice-based schemes, e.g., Kyber [94], Saber [85], NewHope [94], Frodo [12], NTRU Prime [63], etc. More specifically, as our schemes use similar constructions as Saber, most of the power- or EM-based SCA shown on Saber are also applicable to Scabbard's schemes. For example, [85] has shown correlation power analysis based SCA on the Toom-Cook-based polynomial multiplication of Saber; similar attacks are possible on the schemes of Scabbard.

Masking [34] or shuffling [57] are utilized to thwart power-based or EM-based SCA. Between these two countermeasures, masking is a provably secure countermeasure and is usually integrated with lattice-based cryptographic algorithms to prevent SCA [30, 75]. Although shuffling is a low-cost countermeasure compared to masking, it has been shown to be insufficient [96] to prevent SCA when shuffling is used alone. Therefore, the overhead reduction in incorporating masking countermeasures into lattice-based KEMs is one of the crucial steps. One way to achieve this is by exploring masking-friendly design elements of the existing lattice-based KEMs.

In masking, secret dependent variables (e.g., x) are split into multiple separate shares (for the first-order masking, x is divided into two shares x_1 and x_2). Then, all the operations of the cryptographic algorithms are performed independently on all the shares. To achieve efficiency, masking LWE/LWR-based KEMs requires two kinds of masking techniques, i) arithmetic masking ($x = x_1 + x_2 \bmod q$) and ii) Boolean masking ($x = x_1 \oplus x_2$). Masked LWE/LWR-based KEMs mainly use the following components: (i) masked polynomial arithmetic (modular addition, subtraction, and multiplication), (ii) masked compression, (iii) masked message decoding and encoding function, (iv) masked CBD, (v) masked Keccak (used in SHA3-512 and SHAKE-128), and (vi) masked ciphertext comparison. From all the components, (ii) Masked compression, (iii) masked message decoding and encoding function, (iv) masked CBD, and (vi) masked ciphertext comparison require either arithmetic to Boolean (A2B) conversion or Boolean to arithmetic (B2A) conversion. A2B or B2A conversions are one of the performance hefty operations introduced solely due to masking, making the masked components that use them expensive in terms of performance. However, this performance cost heavily depends on the parameters of the LWE/LWR-based KEMs. For example, the modulus q is usually chosen to be prime for LWE-based schemes for being able to use NTT, whereas it is mostly a power-of-2 for LWR-based schemes. A2B and B2A conversions are much cheaper for a power-of-2 modulus compared to a prime modulus with the same bit length. Also, the smaller parameters in our schemes reduce the performance overhead of masking components. More elaborated observations regarding the design choices of the schemes of Scabbard and their effect on masking have been shown in [77]. We have presented some of these results in Appendix A. Overall, all three schemes of Scabbard outperform Kyber on the Cortex-M4 platform when masking countermeasures are integrated.

Masking LWE/LWR-based KEMs can prevent some of the fault-injection attacks (FIA), such as safe-error attacks [22]. However, several FIA have been proposed on masked implementation of LWE/LWR-based KEMs [42, 58, 74, 84, 92, 103]. In fact, [74] demonstrates that masking introduces new attack surfaces for the FIA in the LWE/LWR-based KEMs. The FIA in the context of LWE/LWR-based schemes can be primarily divided into two categories: (i) ineffective fault attacks and (ii) FIA at the ciphertext comparison. In ineffective fault attacks, the secret data-dependent behavioral changes of the decapsulation procedure of the KEMs upon fault injection on a specific variable leak information regarding the secret key. Basically, in these attacks, injected fault changes the value of the targeted variable, which causes decapsulation failure for some values of the targeted variable. For the other values of the targeted variable, the injected fault doesn't affect the final outcome *i.e.* decapsulation success. This phenomenon leaks information regarding the targeted variable's value, which depends on the secret key. Some examples of such fault attacks are [42, 58, 74, 92]. In the FIA at the ciphertext comparison [84, 103], the last equality checking in the decapsulation procedure between re-encrypted ciphertext and received public ciphertext is bypassed. Fundamentally, this removes the Fujisaki-Okamoto transform *i.e* changes a CCA-secure KEM scheme to a CPA-secure KEX scheme. It forces the decapsulation to succeed even in cases where decapsulation would have failed in the normal scenario. Therefore, the adversary can retrieve the long-term secret key from the decapsulation process with the help of specially crafted input ciphertexts.

Scabbard's schemes are also vulnerable to the fault-injection attacks discussed here. Recently, some works [21, 95] have proposed detection-based countermeasures against FIA on LWE/LWR-based KEMs. These countermeasures can be integrated into Scabbard with small adjustments. However, further research is needed to verify the effectiveness and cost of these countermeasures on Scabbard.

8 CONCLUSION

We provide a suite of three LWR-based KEMs by exploring possible design choices and the parameter set. Our study improves the state-of-the-art lattice-based post-quantum cryptography in aspects of software and hardware implementations. We show that the choice of design primitives heavily affects the scheme's efficiency on the software and hardware platforms. In fact, the design choices of a lattice-based KEM also affect the performance overhead of the scheme's secure implementations. The work in [77] experimentally demonstrated that the schemes of Scabbard outperform Kyber on the Cortex-m4 platform when side-channel countermeasure masking is integrated. In this work, we consider implementation aspects during the design of a scheme, which results in a more efficient scheme while providing a similar level of security. Our result opens a new research direction for LWR-based lightweight secure PQC KEMs, which can be extended to LWE-based KEMs. In fact, a new lightweight MLWE-based scheme, Rudraksh [76], has been proposed by performing a similar module-space exploration strategy on LWE-based primitives. We believe this research will benefit other LWE-/LWR-based primitives such as lattice-based digital signature schemes, lightweight schemes, group-key exchange schemes, etc. We have left these as potential future work.

ACKNOWLEDGMENTS

This work was partially supported by Horizon 2020 ERC Advanced Grant (101020005 Belfort), CyberSecurity Research Flanders with reference number VR20192203, BE QCI: Belgian-QCI (3E230370) (see beqci.eu), Intel Corporation, Secure Implementation of Post-Quantum Cryptosystems (SECPQC) DST-India and BELSPO.

REFERENCES

[1] Amin Abdulrahman, Jiun-Peng Chen, Yu-Jia Chen, Vincent Hwang, Matthias J. Kannwischer, and Bo-Yin Yang. 2022. Multi-moduli NTTs for Saber on Cortex-M3 and Cortex-M4. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022, 1 (2022), 127–151. https://doi.org/10.46586/TCHES.V2022.I1.127-151

[2] Aikata Aikata, Ahmet Can Mert, Malik Imran, Samuel Pagliarini, and Sujoy Sinha Roy. 2023. KaLi: A Crystal for Post-Quantum Security Using Kyber and Dilithium. *IEEE Transactions on Circuits and Systems I: Regular Papers* 70, 2 (2023), 747–758. https://doi.org/10.1109/TCSI.2022.3219555

[3] M. Ajtai. 1996. Generating Hard Instances of Lattice Problems (Extended Abstract). In *In Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing*. ACM, 99–108.

[4] Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. 2022. Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process. Online. Accessed 26th January, 2024. https://nvlpubs.nist.gov/nistpubs/ir/2022/NIST.IR.8413-upd1.pdf

[5] Martin R. Albrecht, Benjamin R. Curtis, Amit Deo, Alex Davidson, Rachel Player, Eamonn W. Postlethwaite, Fernando Virdia, and Thomas Wunderer. 2018. Estimate All the {LWE, NTRU} Schemes!. In *Security and Cryptography for Networks - 11th International Conference, SCN 2018, Amalfi, Italy, September 5-7, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11035)*, Dario Catalano and Roberto De Prisco (Eds.). Springer, 351–367. https://doi.org/10.1007/978-3-319-98113-0_19

[6] Martin R. Albrecht, Rachel Player, and Sam Scott. 2015. On the concrete hardness of Learning with Errors. Cryptology ePrint Archive, Report 2015/046. https://eprint.iacr.org/2015/046.

[7] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. 2016. Post-quantum Key Exchange - A New Hope. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, Thorsten Holz and Stefan Savage (Eds.). USENIX Association, 327–343. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/alkim

[8] Jacob Alperin-Sheriff and Daniel Apon. 2016. Dimension-Preserving Reductions from LWE to LWR. *IACR Cryptol. ePrint Arch.* (2016), 589. http://eprint.iacr.org/2016/589

[9] Joël Alwen, Stephan Krenn, Krzysztof Pietrzak, and Daniel Wichs. 2013. Learning with Rounding, Revisited - New Reduction, Properties and Applications. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I (Lecture Notes in Computer Science, Vol. 8042)*, Ran Canetti and Juan A. Garay (Eds.). Springer, 57–74. https://doi.org/10.1007/978-3-642-40041-4_4

[10] Dorian Amiet, Andreas Curiger, Lukas Leuenberger, and Paul Zbinden. 2020. Defeating NewHope with a Single Trace. Cryptology ePrint Archive, Report 2020/368. https://ia.cr/2020/368.

[11] Jean-Philippe Aumasson, Daniel J. Bernstein, Ward Beullens, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Andreas H ulsing, Panos Kampanakis, Stefan Kölbl, Tanja Lange, Florian Mendel Martin M. Lauridsen, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, and Bas Westerbaan. 2018. SPHINCS+ Submission to the NIST post-quantum project, v.3.1. https://sphincs.org/data/sphincs+-r3.1-specification.pdf [Online; accessed 14-March-2024].

[12] Furkan Aydin, Aydin Aysu, Mohit Tiwari, Andreas Gerstlauer, and Michael Orshansky. 2021. Horizontal Side-Channel Vulnerabilities of Post-Quantum Key Exchange and Encapsulation Protocols. *ACM Trans. Embed. Comput. Syst.* 20, 6 (2021), 110:1–110:22. https://doi.org/10.1145/3476799

[13] Abhishek Banerjee, Chris Peikert, and Alon Rosen. 2012. Pseudorandom Functions and Lattices. In *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7237)*, David Pointcheval and Thomas Johansson (Eds.). Springer, 719–737. https://doi.org/10.1007/978-3-642-29011-4_42

[14] Andrea Basso, Jose Maria Bermudo Mera, Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, Michiel Van Beirendonck, and Frederik Vercauteren. 2020. SABER: Mod-LWR based KEM (Round 3 Submission). https://www.esat.kuleuven.be/cosic/pqcrypto/saber/files/saberspecround3.pdf. https://www.esat.kuleuven.be/cosic/pqcrypto/saber/files/saberspecround3.pdf [Online; accessed 3-July-2021].

[15] Anja Becker, Léo Ducas, Nicolas Gama, and Thijs Laarhoven. 2016. New directions in nearest neighbor searching with applications to lattice sieving. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, Robert Krauthgamer (Ed.). SIAM, 10–24. https://doi.org/10.1137/1.9781611974331.CH2

[16] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. [n. d.]. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. https://eprint.iacr.org/2021/986.

[17] Michiel Van Beirendonck, Jan-Pieter D'Anvers, Angshuman Karmakar, Josep Balasch, and Ingrid Verbauwhede. 2020. A Side-Channel Resistant Implementation of SABER. Cryptology ePrint Archive, Report 2020/733. https://ia.cr/2020/733.

[18] Jose Maria Bermudo Mera, Angshuman Karmakar, Suparna Kundu, and Ingrid Verbauwhede. 2021. Scabbard: a suite of efficient learning with rounding key-encapsulation mechanisms. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021, 4 (Aug. 2021), 474–509. https://doi.org/10.46586/tches.v2021.i4.474-509

[19] Daniel J. Bernstein, Karthikeyan Bhargavan, Shivam Bhasin, Anupam Chattopadhyay, Tee Kiah Chia, Matthias J. Kannwischer, Franziskus Kiefer, Thales Paiva, Prasanna Ravi, and Goutam Tamvada. 2024. KyberSlash: Exploiting secret-dependent division timings in Kyber implementations. *IACR Cryptol. ePrint Arch.* (2024), 1049. https://eprint.iacr.org/2024/1049

[20] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. 2016. NTRU Prime: reducing attack surface at low cost. Cryptology ePrint Archive, Report 2016/461. https://eprint.iacr.org/2016/461.

[21] Pierre-Augustin Berthet, Cédric Tavernier, Jean-Luc Danger, and Laurent Sauvage. 2023. Quasi-linear Masking to Protect Kyber against both SCA and FIA. *IACR Cryptol. ePrint Arch.* (2023), 1220. https://eprint.iacr.org/2023/1220

[22] Luk Bettale, Simon Montoya, and Guénaël Renault. 2021. Safe-Error Analysis of Post-Quantum Cryptography Mechanisms - Short Paper-. In *18th Workshop on Fault Detection and Tolerance in Cryptography, FDTC 2021, Milan, Italy, September 17, 2021.* IEEE, 39–44. https://doi.org/10.1109/FDTC53659.2021.00015

[23] Sauvik Bhattacharya, Oscar Garcia-Morchon, Thijs Laarhoven, Ronald Rietman, Markku-Juhani O. Saarinen, Ludo Tolhuizen, and Zhenfei Zhang. 2018. Round5: KEM and PKE based on GLWR. Cryptology ePrint Archive, Report 2018/725. https://eprint.iacr.org/2018/725.

[24] Mojtaba Bisheh-Niasar, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. 2021. High-Speed NTT-based Polynomial Multiplication Accelerator for CRYSTALS-Kyber Post-Quantum Cryptography. Cryptology ePrint Archive, Paper 2021/563. https://eprint.iacr.org/2021/563 https://eprint.iacr.org/2021/563.

[25] Andrej Bogdanov, Siyao Guo, Daniel Masny, Silas Richelson, and Alon Rosen. 2016. On the Hardness of Learning with Rounding over Small Modulus. In *Theory of Cryptography - 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9562)*, Eyal Kushilevitz and Tal Malkin (Eds.). Springer, 209–224. https://doi.org/10.1007/978-3-662-49096-9_9

[26] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. 1997. On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract). In *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding (Lecture Notes in Computer Science, Vol. 1233)*, Walter Fumy (Ed.). Springer, 37–51. https://doi.org/10.1007/3-540-69053-0_4

[27] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2017. CRYSTALS – Kyber: a CCA-secure module-lattice-based KEM. Cryptology ePrint Archive, Report 2017/634. https://ia.cr/2017/634.

[28] Joppe W. Bos, Olivier Bronchain, Frank Custers, Joost Renes, Denise Verbakel, and Christine van Vredendaal. 2023. Enabling FrodoKEM on Embedded Devices. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2023, 3 (2023), 74–96. https://doi.org/10.46586/TCHES.V2023.I3.74-96

[29] Joppe W. Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. 2016. Frodo: Take off the Ring! Practical, Quantum-Secure Key Exchange from LWE. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 1006–1018. https://doi.org/10.1145/2976749.2978425

[30] Olivier Bronchain and Gaëtan Cassiers. 2022. Bitslicing Arithmetic/Boolean Masking Conversions for Fun and Profit with Application to Lattice-Based KEMs. Cryptology ePrint Archive, Report 2022/158. https://ia.cr/2022/158.

[31] Joan Bruna, Oded Regev, Min Jae Song, and Yi Tang. 2021. Continuous LWE. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing* (Virtual, Italy) *(STOC 2021)*. Association for Computing Machinery, New York, NY, USA, 694–707. https://doi.org/10.1145/3406325.3451000

[32] BSI. 2023. Cryptographic Mechanisms: Recommendations and Key Lengths. https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TG02102/BSI-TR-02102-1.pdf?__blob=publicationFile&v=10. https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TG02102/BSI-TR-02102-1.pdf?__blob=publicationFile&v=10

[33] Johannes Buchmann, Florian Göpfert, Tim Güneysu, Tobias Oder, and Thomas Pöppelmann. 2016. High-Performance and Lightweight Lattice-Based Public-Key Encryption. In *Proceedings of the 2nd ACM International Workshop on IoT Privacy, Trust, and Security, IoTPTS@AsiaCCS, Xi'an, China, May 30, 2016*, Richard Chow and Gökay Saldamli (Eds.). ACM, 2–9. https://doi.org/10.1145/2899007.2899011

[34] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. 1999. Towards Sound Approaches to Counteract Power-Analysis Attacks. In *Advances in Cryptology — CRYPTO' 99*, Michael Wiener (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 398–412.

[35] Jung Hee Cheon, Hyeongmin Choe, Dongyeon Hong, and MinJune Yi. 2023. SMAUG: Pushing Lattice-based Key Encapsulation Mechanisms to the Limits. *IACR Cryptol. ePrint Arch.* (2023), 739. https://eprint.iacr.org/2023/739

[36] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. 2020. NTT Multiplication for NTT-unfriendly Rings. Cryptology ePrint Archive, Report 2020/1397. https://eprint.iacr.org/2020/1397.

[37] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. 2021. NTT Multiplication for NTT-unfriendly Rings: New Speed Records for Saber and NTRU on Cortex-M4 and AVX2. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021, 2 (Feb. 2021), 159–188.

[38] Dana Dachman-Soled, Léo Ducas, Huijing Gong, and Mélissa Rossi. 2020. LWE with Side Information: Attacks and Concrete Security Estimation. Cryptology ePrint Archive, Report 2020/292. https://eprint.iacr.org/2020/292.

[39] Viet B. Dang, Farnoud Farahmand, Michal Andrzejczak, and Kris Gaj. 2019. Implementing and Benchmarking Three Lattice-Based Post-Quantum Cryptography Algorithms Using Software/Hardware Codesign. In *2019 International Conference on Field-Programmable Technology (ICFPT)*. 206–214. https://doi.org/10.1109/ICFPT47387.2019.00032

[40] Viet Ba Dang, Kamyar Mohajerani, and Kris Gaj. 2023. High-Speed Hardware Architectures and FPGA Benchmarking of CRYSTALS-Kyber, NTRU, and Saber. *IEEE Trans. Comput.* 72, 2 (2023), 306–320. https://doi.org/10.1109/TC.2022.3222954

[41] Jan-Pieter D'Anvers, Marcel Tiepelt, Frederik Vercauteren, and Ingrid Verbauwhede. 2019. Timing attacks on Error Correcting Codes in Post-Quantum Schemes. Cryptology ePrint Archive, Report 2019/292. https://eprint.iacr.org/2019/292.

[42] Jeroen Delvaux. 2022. Roulette: A Diverse Family of Feasible Fault Attacks on Masked Kyber. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022, 4 (2022), 637–660. https://doi.org/10.46586/tches.v2022.i4.637-660

[43] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2018. CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018, 1 (Feb. 2018), 238–268. https://doi.org/10.13154/tches.v2018.i1.238-268

[44] Morris Dworkin. 2015. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. https://doi.org/10.6028/NIST.FIPS.202

[45] Shahriar Ebrahimi and Siavash Bayat-Sarmadi. 2020. Lightweight and Fault-Resilient Implementations of Binary Ring-LWE for IoT Devices. *IEEE Internet of Things Journal* 7, 8 (2020), 6970–6978. https://doi.org/10.1109/JIOT.2020.2979318

[46] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. 2018. Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU. https://falcon-sign.info/ [Online; accessed 14-March-2024].

[47] Eiichiro Fujisaki and Tatsuaki Okamoto. 2013. Secure Integration of Asymmetric and Symmetric Encryption Schemes. *J. Cryptol.* 26, 1 (2013), 80–101. https://doi.org/10.1007/s00145-011-9114-1

[48] Archisman Ghosh, Jose Maria Bermudo Mera, Angshuman Karmakar, Debayan Das, Santosh Ghosh, Ingrid Verbauwhede, and Shreyas Sen. 2022. A 334uW 0.158mm2 Saber Learning with Rounding based Post-Quantum Crypto Accelerator. In *IEEE Custom Integrated Circuits Conference, CICC 2022, Newport Beach, CA, USA, April 24-27, 2022*. IEEE, 1–2. https://doi.org/10.1109/CICC53496.2022.9772859

[49] Archisman Ghosh, Jose Maria Bermudo Mera, Angshuman Karmakar, Debayan Das, Santosh Ghosh, Ingrid Verbauwhede, and Shreyas Sen. 2023. A 334 μW 0.158 mm^2 ASIC for Post-Quantum Key-Encapsulation Mechanism Saber With Low-Latency Striding Toom–Cook Multiplication. *IEEE Journal of Solid-State Circuits* 58, 8 (2023), 2383–2398. https://doi.org/10.1109/JSSC.2023.3253425

[50] Archisman Ghosh, Jose Maria Bermudo Mera, Angshuman Karmakar, Debayan Das, Santosh Ghosh, Ingrid Verbauwhede, and Shreyas Sen. 2023. A 334 μW 0.158 mm^2 ASIC for Post-Quantum Key-Encapsulation Mechanism Saber With Low-Latency Striding Toom-Cook Multiplication. *IEEE J. Solid State Circuits* 58, 8 (2023), 2383–2398. https://doi.org/10.1109/JSSC.2023.3253425

[51] Aurelien Greuet, Simon Montoya, and Guenael Renault. 2020. Attack on LAC Key Exchange in Misuse Situation. Cryptology ePrint Archive, Report 2020/063. https://eprint.iacr.org/2020/063.

[52] Lov K. Grover. 1996. A Fast Quantum Mechanical Algorithm for Database Search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, Gary L. Miller (Ed.). ACM, 212–219. https://doi.org/10.1145/237814.237866

[53] GSMA. 2024. Post Quantum Cryptography – Guidelines for Telecom Use Cases. https://www.gsma.com/newsroom/wp-content/uploads//PQ.03-Post-Quantum-Cryptography-Guidelines-for-Telecom-Use-v1.0.pdf. https://www.gsma.com/newsroom/wp-content/uploads//PQ.03-Post-Quantum-Cryptography-Guidelines-for-Telecom-Use-v1.0.pdf

[54] Qian Guo, Thomas Johansson, and Alexander Nilsson. 2020. A key-recovery timing attack on post-quantum primitives using the Fujisaki-Okamoto transformation and its application on FrodoKEM. Cryptology ePrint Archive, Report 2020/743. https://ia.cr/2020/743.

[55] Qian Guo, Thomas Johansson, and Jing Yang. 2019. A Novel CCA Attack using Decryption Errors against LAC. Cryptology ePrint Archive, Report 2019/1308. https://eprint.iacr.org/2019/1308.

[56] Daniel Heinz, Matthias J. Kannwischer, Georg Land, Thomas Pöppelmann, Peter Schwabe, and Daan Sprenkels. 2022. First-Order Masked Kyber on ARM Cortex-M4. *IACR Cryptol. ePrint Arch.* (2022), 58. https://eprint.iacr.org/2022/058

[57] Christoph Herbst, Elisabeth Oswald, and Stefan Mangard. 2006. An AES Smart Card Implementation Resistant to Power Analysis Attacks. In *Applied Cryptography and Network Security, 4th International Conference, ACNS 2006, Singapore, June 6-9, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3989)*, Jianying Zhou, Moti Yung, and Feng Bao (Eds.). 239–252. https://doi.org/10.1007/11767480_16

[58] Julius Hermelink, Peter Pessl, and Thomas Pöppelmann. 2021. Fault-Enabled Chosen-Ciphertext Attacks on Kyber. In *Progress in Cryptology - INDOCRYPT 2021 - 22nd International Conference on Cryptology in India, Jaipur, India, December 12-15, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 13143)*, Avishek Adhikari, Ralf Küsters, and Bart Preneel (Eds.). Springer, 311–334. https://doi.org/10.1007/978-3-030-92518-5_15

[59] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. 1998. NTRU: A Ring-Based Public Key Cryptosystem. In *Algorithmic Number Theory, Third International Symposium, ANTS-III, Portland, Oregon, USA, June 21-25, 1998, Proceedings (Lecture Notes in Computer Science, Vol. 1423)*, Joe Buhler (Ed.). Springer, 267–288. https://doi.org/10.1007/BFb0054868

[60] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. 2017. A Modular Analysis of the Fujisaki-Okamoto Transformation. In *Theory of Cryptography - 15th International Conference, TCC 2017, Baltimore, MD, USA, November 12-15, 2017, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10677)*, Yael Kalai and Leonid Reyzin (Eds.). Springer, 341–371. https://doi.org/10.1007/978-3-319-70500-2_12

[61] James Howe, Tobias Oder, Markus Krausz, and Tim Güneysu. 2018. Standard Lattice-Based Key Encapsulation on Embedded Devices. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018, 3 (Aug. 2018), 372–393. https://doi.org/10.13154/tches.v2018.i3.372-393

[62] Junhao Huang, Jipeng Zhang, Haosong Zhao, Zhe Liu, Ray C. C. Cheung, Çetin Kaya Koç, and Donglong Chen. 2022. Improved Plantard Arithmetic for Lattice-based Cryptography. Cryptology ePrint Archive, Paper 2022/956. https://eprint.iacr.org/2022/956 https://eprint.iacr.org/2022/956.

[63] Wei-Lun Huang, Jiun-Peng Chen, and Bo-Yin Yang. 2019. Power Analysis on NTRU Prime. Cryptology ePrint Archive, Report 2019/100. https://ia.cr/2019/100.

[64] Haodong Jiang, Zhenfeng Zhang, Long Chen, Hong Wang, and Zhi Ma. 2017. Post-quantum IND-CCA-secure KEM without Additional Hash. *IACR Cryptol. ePrint Arch.* 2017 (2017), 1096. http://eprint.iacr.org/2017/1096

[65] Matthias J. Kannwischer, Joost Rijneveld, and Peter Schwabe. 2018. Faster multiplication in $\mathbb{Z}_{2^m}[x]$ on Cortex-M4 to speed up NIST PQC candidates. Cryptology ePrint Archive, Report 2018/1018. https://eprint.iacr.org/2018/1018,.

[66] Matthias J. Kannwischer, Joost Rijneveld, and Peter Schwabe. 2018. Faster multiplication in $\mathbb{Z}_{2^m}[x]$ on Cortex-M4 to speed up NIST PQC candidates. Cryptology ePrint Archive, Report 2018/1018. https://ia.cr/2018/1018.

[67] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. [n. d.]. PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4.

[68] Angshuman Karmakar, Jose Maria Bermudo Mera, Sujoy Sinha Roy, and Ingrid Verbauwhede. 2018. Saber on ARM CCA-secure module lattice-based key encapsulation on ARM. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018, 3 (2018), 243–266. https://doi.org/10.13154/tches.v2018.i3.243-266

[69] Jonghyun Kim and Jong Hwan Park. 2023. NTRU+: Compact Construction of NTRU Using Simple Encoding Method. *IEEE Transactions on Information Forensics and Security* 18 (2023), 4760–4774. https://doi.org/10.1109/TIFS.2023.3299172

[70] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings (Lecture Notes in Computer Science, Vol. 1109)*, Neal Koblitz (Ed.). Springer, 104–113. https://doi.org/10.1007/3-540-68697-5_9

[71] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential Power Analysis. In *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1666)*, Michael J. Wiener (Ed.). Springer, 388–397. https://doi.org/10.1007/3-540-48405-1_25

[72] François Koeune and François-Xavier Standaert. 2004. A Tutorial on Physical Security and Side-Channel Attacks. In *Foundations of Security Analysis and Design III, FOSAD 2004/2005 Tutorial Lectures (Lecture Notes in Computer Science, Vol. 3655)*, Alessandro Aldini, Roberto Gorrieri, and Fabio Martinelli (Eds.). Springer, 78–108. https://doi.org/10.1007/11554578_3

[73] KpqC. [n. d.]. Korean PQC competition. https://www.kpqc.or.kr/competition.html. https://www.kpqc.or.kr/competition.html [Online; accessed 10-January-2024].

[74] Suparna Kundu, Siddhartha Chowdhury, Sayandeep Saha, Angshuman Karmakar, Debdeep Mukhopadhyay, and Ingrid Verbauwhede. 2024. Carry Your Fault: A Fault Propagation Attack on Side-Channel Protected LWE-based KEM. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2024, 2 (2024), 844–869. https://doi.org/10.46586/TCHES.V2024.I2.844-869

[75] Suparna Kundu, Jan-Pieter D'Anvers, Michiel Van Beirendonck, Angshuman Karmakar, and Ingrid Verbauwhede. 2022. Higher-Order Masked Saber. In *Security and Cryptography for Networks*, Clemente Galdi and Stanislaw Jarecki (Eds.). Springer International Publishing, Cham, 93–116.

[76] Suparna Kundu, Archisman Ghosh, Angshuman Karmakar, Shreyas Sen, and Ingrid Verbauwhede. 2024. Rudraksh: A compact and lightweight post-quantum key-encapsulation mechanism. Cryptology ePrint Archive, Paper 2024/1170. https://eprint.iacr.org/2024/1170 https://eprint.iacr.org/2024/1170.

[77] Suparna Kundu, Angshuman Karmakar, and Ingrid Verbauwhede. 2023. On the Masking-Friendly Designs for Post-quantum Cryptography. In *Security, Privacy, and Applied Cryptography Engineering - 13th International Conference, SPACE 2023, Roorkee, India, December 14-17, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 14412)*, Francesco Regazzoni, Bodhisatwa Mazumdar, and Sri Parameswaran (Eds.). Springer, 162–184. https://doi.org/10.1007/978-3-031-51583-5_10

[78] Adeline Langlois and Damien Stehlé. 2015. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography* 75, 3 (01 Jun 2015), 565–599. https://doi.org/10.1007/s10623-014-9938-4

[79] Patrick Longa and Michael Naehrig. 2016. Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography. In *Cryptology and Network Security*, Sara Foresti and Giuseppe Persiano (Eds.). Springer International Publishing, Cham, 124–139.

[80] Xianhui Lu, Yamin Liu, Zhenfei Zhang, Dingding Jia, Haiyang Xue, Jingnan He, Bao Li, and Kunpeng Wang. 2018. LAC: Practical Ring-LWE Based Public-Key Encryption with Byte-Level Modulus. Cryptology ePrint Archive, Report 2018/1009. https://eprint.iacr.org/2018/1009.

[81] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. 2010. On Ideal Lattices and Learning with Errors over Rings. In *Advances in Cryptology – EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 – June 3, 2010. Proceedings*, Henri Gilbert (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–23. https://doi.org/10.1007/978-3-642-13190-5_1

[82] Jose Maria Bermudo Mera, Angshuman Karmakar, and Ingrid Verbauwhede. 2020. Time-memory trade-off in Toom-Cook multiplication: an application to module-lattice based cryptography. *IACR Cryptol. ePrint Arch.* 2020 (2020), 268. https://eprint.iacr.org/2020/268

[83] Jose Maria Bermudo Mera, Furkan Turan, Angshuman Karmakar, Sujoy Sinha Roy, and Ingrid Verbauwhede. 2020. Compact domain-specific co-processor for accelerating module lattice-based KEM. In *57th ACM/IEEE Design Automation Conference, DAC 2020, San Francisco, CA, USA, July 20-24, 2020.* IEEE, 1–6. https://doi.org/10.1109/DAC18072.2020.9218727

[84] Puja Mondal, Suparna Kundu, Sarani Bhattacharya, Angshuman Karmakar, and Ingrid Verbauwhede. 2024. A Practical Key-Recovery Attack on LWE-Based Key-Encapsulation Mechanism Schemes Using Rowhammer. In *Applied Cryptography and Network Security - 22nd International Conference, ACNS 2024, Abu Dhabi, United Arab Emirates, March 5-8, 2024, Proceedings, Part III (Lecture Notes in Computer Science, Vol. 14585)*, Christina Pöpper and Lejla Batina (Eds.). Springer, 271–300. https://doi.org/10.1007/978-3-031-54776-8_11

[85] Catinca Mujdei, Lennert Wouters, Angshuman Karmakar, Arthur Beckers, Jose Maria Bermudo Mera, and Ingrid Verbauwhede. 2022. Side-Channel Analysis of Lattice-Based Post-Quantum Cryptography: Exploiting Polynomial Multiplication. *ACM Trans. Embed. Comput. Syst.* (nov 2022). https://doi.org/10.1145/3569420 Just Accepted.

[86] NIST. 2009. Digital Signature Standard (DSS). https://csrc.nist.gov/files/pubs/fips/186-3/final/docs/fips_186-3.pdf

[87] NIST. 2013. Digital Signature Standard (DSS). https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf

[88] NIST. 2013. NIST Curves. Online. Accessed 15th March, 2024. https://csrc.nist.gov/csrc/media/events/ispab-december-2013-meeting/documents/nist_elliptic-curves.pdf

[89] NIST. 2023. Digital Signature Standard (DSS). https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-5.pdf

[90] Seunghwan Park, Chi-Gon Jung, Aesun Park, Joongeun Choi, and Honggoo Kang. 2022. TiGER: Tiny bandwidth key encapsulation mechanism for easy miGration based on RLWE(R). *IACR Cryptol. ePrint Arch.* (2022), 1651. https://eprint.iacr.org/2022/1651

[91] Bo-Yuan Peng, Adrian Marotzke, Ming-Han Tsai, Bo-Yin Yang, and Ho-Lin Chen. 2022. Streamlined NTRU Prime on FPGA. *Journal of Cryptographic Engineering* 13 (2022), 167 – 186. https://api.semanticscholar.org/CorpusID:243990500

[92] Peter Pessl and Lukas Prokop. 2021. Fault Attacks on CCA-secure Lattice KEMs. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021, 2 (2021), 37–60. https://doi.org/10.46586/tches.v2021.i2.37-60

[93] John Proos and Christof Zalka. 2003. Shor's discrete logarithm quantum algorithm for elliptic curves. *Quantum Inf. Comput.* 3, 4 (2003), 317–344. https://doi.org/10.26421/QIC3.4-3

[94] Prasanna Ravi, Shivam Bhasin, Sujoy Sinha Roy, and Anupam Chattopadhyay. 2020. Drop by Drop you break the rock - Exploiting generic vulnerabilities in Lattice-based PKE/KEMs using EM-based Physical Attacks. Cryptology ePrint Archive, Report 2020/549. https://ia.cr/2020/549.

[95] Prasanna Ravi, Anupam Chattopadhyay, Jan-Pieter D'Anvers, and Anubhab Baksi. 2024. Side-channel and Fault-injection attacks over Lattice-based Post-quantum Schemes (Kyber, Dilithium): Survey and New Results. *ACM Trans. Embed. Comput. Syst.* 23, 2 (2024), 35:1–35:54. https://doi.org/10.1145/3603170

[96] Prasanna Ravi, Thales Paiva, Dirmanto Jap, Jan-Pieter D'Anvers, and Shivam Bhasin. 2024. Defeating Low-Cost Countermeasures against Side-Channel Attacks in Lattice-based Encryption A Case Study on Crystals-Kyber. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2024, 2 (2024), 795–818. https://doi.org/10.46586/TCHES.V2024.I2.795-818

[97] Prasanna Ravi, Debapriya Basu Roy, Shivam Bhasin, Anupam Chattopadhyay, and Debdeep Mukhopadhyay. 2019. Number "Not Used" Once - Practical Fault Attack on pqm4 Implementations of NIST Candidates. In *Constructive Side-Channel Analysis and Secure Design - 10th International Workshop, COSADE 2019, Darmstadt, Germany, April 3-5, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11421)*, Ilia Polian and Marc Stöttinger (Eds.). Springer, 232–250. https://doi.org/10.1007/978-3-030-16350-1_13

[98] Oded Regev. 2004. *New Lattice-based Cryptographic Constructions*. Vol. 51-6. ACM, New York, NY, USA, 899–942. https://doi.org/10.1145/1039488.1039490

[99] Miruna Rosca, Damien Stehlé, and Alexandre Wallet. 2018. On the Ring-LWE and Polynomial-LWE Problems. In *Advances in Cryptology – EUROCRYPT 2018*, Jesper Buus Nielsen and Vincent Rijmen (Eds.). Springer International Publishing, Cham, 146–173.

[100] Sujoy Sinha Roy and Andrea Basso. 2020. High-speed Instruction-set Coprocessor for Lattice-based Key Encapsulation Mechanism: Saber in Hardware. *IACR Cryptol. ePrint Arch.* 2020 (2020), 434. https://eprint.iacr.org/2020/434

[101] Peter W. Shor. 1994. Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*. IEEE Computer Society, 124–134. https://doi.org/10.1109/SFCS.1994.365700

[102] Keccak Team. [n. d.]. Keccak in VHDL: High-speed core. Online. Accessed 29th February, 2024. https://keccak.team/hardware.html

[103] Keita Xagawa, Akira Ito, Rei Ueno, Junko Takahashi, and Naofumi Homma. 2021. Fault-Injection Attacks Against NIST's Post-Quantum Cryptography Round 3 KEM Candidates. In *Advances in Cryptology - ASIACRYPT 2021 - 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6-10, 2021, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 13091)*, Mehdi Tibouchi and Huaxiong Wang (Eds.). Springer, 33–61. https://doi.org/10.1007/978-3-030-92075-3_2

A PERFORMANCE OF MASKED SCABBARD

Kundu et al. [77] integrated masking countermeasures on the medium security version (NIST-3) of all the schemes of Scabbard. It also proposes proof-of-concept implementations on the ARM Cortex-M4 platform using the PQM4 framework [67]. The test vector leakage assessment hasn't been performed and is left as future work. Table 11, 12, and 13 present performance results of masked Florete, Espada, and Sable, respectively. In Table 14, masked implementations of Scabbard have been compared with the state-of-the-art implementations of LWE/LWR-based KEMs, including Kyber.

Table 11. Performance of components of Florete on Cortex-M4 [77]

	x1000 clock cycles						
Order	Unmask	1st		2nd		3rd	
Florete CCA-KEM-Decapsulation	954	2,621	(2.74x)	4,844	(5.07x)	7,395	(7.75x)
CPA-PKE-Decryption	248	615	(2.47x)	1,107	(4.46x)	1,651	(6.65x)
Polynomial arithmetic	241	461	(1.91x)	690	(2.86x)	917	(3.80x)
Compression *original_msg*	6	153	(25.50x)	416	(69.33x)	734	(122.33x)
Hash $\mathcal{G}$ (SHA3-512)	13	123	(9.46x)	242	(18.61x)	379	(29.15x)
CPA-PKE-Encryption	554	1,744	(3.14x)	3,354	(6.05x)	5,225	(9.43x)
Secret generation	29	427	(14.72x)	982	(33.86x)	1,663	(57.34x)
XOF (SHAKE-128)	25	245	(9.80x)	484	(19.36x)	756	(30.24x)
CBD (β_1)	4	182	(45.50x)	497	(124.25x)	907	(226.75x)
Polynomial arithmetic *arrange_msg*	524	943	(2.51x)	1,357	(4.52x)	1,783	(6.79x)
Polynomial Comparison		373		1,014		1,778	
Other operations	138	139	(1.00x)	140	(1.01x)	140	(1.01x)

Table 12. Performance of components of Espada on Cortex-M4 [77]

	x1000 clock cycles						
Order	Unmask	1st		2nd		3rd	
Espada CCA-KEM-Decapsulation	2,422	4,335	(1.78x)	6,838	(2.82x)	9,861	(4.07x)
CPA-PKE-Decryption	70	137	(1.95x)	230	(3.28x)	324	(4.62x)
Polynomial arithmetic	69	116	(1.68x)	170	(2.46x)	225	(3.26x)
Compression *original_msg*	0.4	20	(50.00x)	60	(150.00x)	99	(247.50x)
Hash $\mathcal{G}$ (SHA3-512)	13	123	(9.46x)	243	(18.69x)	379	(29.15x)
CPA-PKE-Encryption	2,215	3,950	(1.78x)	6,240	(2.81x)	9,031	(4.07x)
Secret generation	57	748	(13.12x)	1,650	(28.94x)	3,009	(52.78x)
XOF (SHAKE-128)	51	489	(9.58x)	968	(18.98x)	1,510	(29.60x)
CBD (β_3)	6	259	(43.16x)	681	(113.50x)	1,498	(249.66x)
Polynomial arithmetic *arrange_msg*	2,157	2,865	(1.44x)	3,593	(2.12x)	4,354	(2.79x)
Polynomial Comparison		259		996		1,667	
Other operations	124	124	(1.00x)	124	(1.00x)	126	(1.01x)

Table 13. Performance of components of Sable on Cortex-M4 [77]

	x1000 clock cycles						
Order	Unmask	1st		2nd		3rd	
Sable CCA-KEM-Decapsulation	1,020	2,431	(2.38x)	4,348	(4.26x)	6,480	(6.35x)
CPA-PKE-Decryption	130	291	(2.23x)	510	(3.92x)	745	(5.73x)
Polynomial arithmetic	128	238	(1.85x)	350	(2.73x)	465	(3.63x)
Compression *original_msg*	2	52	(26.00x)	160	(80.00x)	280	(140.00x)
Hash $\mathcal{G}$ (SHA3-512)	13	123	(9.46x)	242	(18.61x)	379	(29.15x)
CPA-PKE-Encryption	764	1,903	(2.49x)	3,482	(4.55x)	5,241	(6.85x)
Secret generation	29	427	(14.72x)	984	(33.93x)	1,666	(57.44x)
XOF (SHAKE-128)	25	245	(9.80x)	484	(19.36x)	756	(30.24x)
CBD (β_1)	4	182	(45.50x)	499	(124.75x)	909	(227.25x)
Polynomial arithmetic *arrange_msg*	734	1,187	(2.00x)	1,640	(3.40x)	2,086	(4.86x)
Polynomial Comparison		287		856		1,488	
Other operations	112	113	(1.00x)	113	(1.00x)	113	(1.00x)

Table 14. Comparing performance of masked Scabbard with the state-of-the-art [77]

Scheme		Performance			# Random numbers		
		(x1000 clock cycles)			(bytes)		
		1st	2nd	3rd	1st	2nd	3rd
Florete	[77]	2,621	4,844	7,395	15,824	52,176	101,280
Espada	[77]	4,335	6,838	9,861	11,496	39,320	85,296
Sable	[77]	**2,431**	**4,348**	**6,480**	12,496	39,152	75,232
Saber	[75]	3,022	5,567	8,649	12,752	43,760	93,664
uSaber	[75]	2,473	4,452	6,947	10,544	36,848	79,840
Kyber	[30]	10,018	16,747	24,709	-	-	-